Pseudo-Boolean Constraint Solving using Stochastic Local Search

This assignment is for 396 students only. 296 students are welcome to do it of course, but only 396 students need to turn it in.

Overview

In this assignment, you will implement the inner loop of the SAT solver demonstrated in lecture. However, we will do a slightly fancier version that understands pseudo-Boolean constraints (PBCs). PBCs are a generalization of clauses: a clause is satisfied if at least one of its literals is true. But a PBC can specify arbitrary minimum and maximum numbers of literals.

So the Imaginarium rule:

> Beagles are small

Corresponds to the implication beagle ⇒ dog, which is the clause ¬beagle ∨ dog, which is equivalent to the PBC:

> At least one of: not beagle | dog

However, the Imaginarium rule:

> Husky and beagle are kinds of dog

Says that any dog has to be exactly one of the subkinds of dog, but that rule only applies to dogs. That doesn't correspond to a single clause, but it does corresponds to a single PBC:

> Exactly one of: husky | beagle | not dog

The same WalkSAT algorithm we discussed in lecture works for PBCs with minor changes:

- You have to change the definition of what it means for a constraint to be satisfied, since there are now both minimum and maximum numbers of literals, whereas for clauses min=1 and there's no max.
- You should really change the greedy flip algorithm to check whether it's trying to increase or decrease the number of true literals in the chosen constraint, and only pick flips that with increase the number of true literals we're below the minimum, and only decrease the number of true literals when we're above the maximum.

Getting started

The system is written to run under the Unity3D game engine. You want version 2019.4.37f1, which you can download here. As of this writing, they look like they may be about to switch to 2019.4.38f1, and that's fine too. If you already have Unity installed and are using some other version, all that matters is that you can run the project under it; use whatever version you want as long as Unity doesn't complain.

When Unity asks you what kind of license to use, you the personal license.

Once you've installed Unity, unzip the assignment and find the folder `Assets/Scenes` and open the file `SATToy.unity` within it. This will launch unity and open the proper scene file. This will launch the Unity editor:
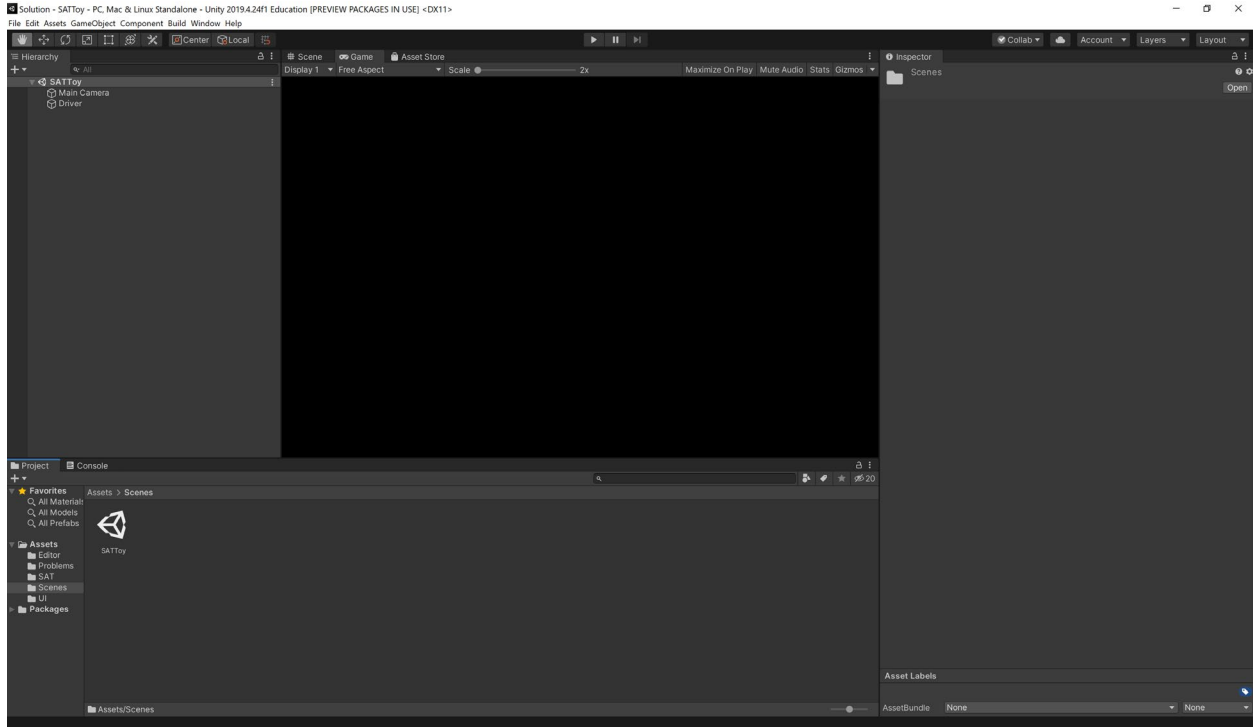

The only parts of this you need to know about are:

- The play button (top of the window, in the center)
  This starts and stops your program.
- The "Console" tab
  This is where error messages (compiler errors and exceptions) will print, as well as any logging you do using the `Debug.Log(string message)` method, which will print strings to this window. So if you want to do the equivalent of a debugging printf, `Debug.Log` is what you want.

You can also install Visual Studio Tools for Unity, which will let you do full source-level debugging (breakpoints, single-stepping, etc.) of your project. As you know, I normally strongly encourage people to do that. But for this assignment, I built a lot of debugging tools into the solver itself. So you should actually be find just debugging with the single-stepping built into the solver, and the use of `Debug.Log()`. But if you start getting a lot of exceptions, you might want to do real debugging.

Running the solver

Just push the play (▶)button at the center top of the screen. That will start the code running. You can stop it at any time by pressing the button again. However, understand that Unity is running in the same thread as your code. So if your code goes into an infinite loop, unity won't ever have a chance to

check the play button. In that case you will need to Force Quit (MacOS) or use the Task Manager (windows) to completely kill unity and restart it.
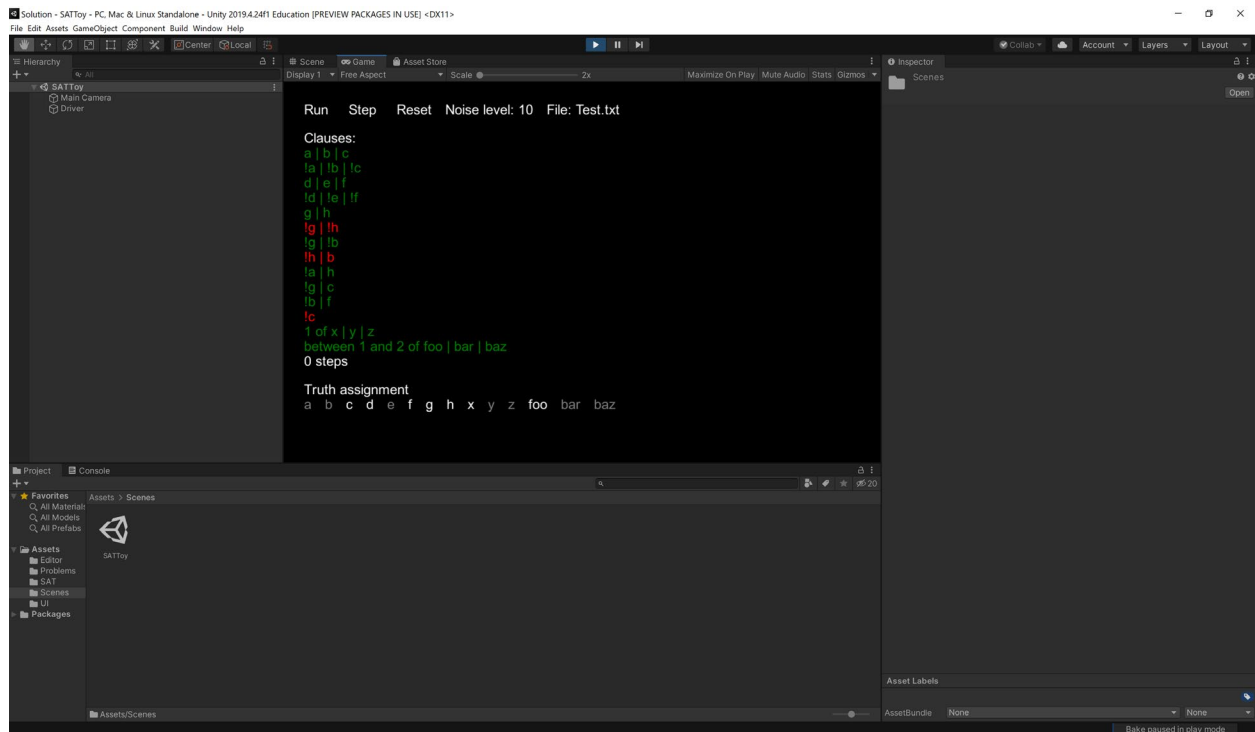
We demoed the solver in the lecture, but here's the summary of the buttons at the top of the screen:

- **Run**
  The solver continually flips literals until it has a solution. When you press it, this becomes a Stop button, which pauses the flipping.
- **Step**
  Does one flip and updates the screen.
- **Reset**
  Changes the truth assignment to a random set of true and false values.
- **Noise level**
  Changes how frequently the solver does completely random flips, as opposed to greedy ones.
- **File:**
  You can type the name of the problem file you want to use here. The problem files are stored in Assets/Problems. We provide you with Test.txt, Creature.txt, and Grid.txt, but we encourage you to make others.

Your job

You need to fill in the following methods in Problem.cs:

- bool **StepOne()**
  This is the inner loop of the WalkSAT algorithm discussed in lecture. Chooses an unsatisfied constraint, then chooses a literal within it to flip. It then calls Flip(), below, to flip its value.

Returns true if there are no remaining unsatisfied clauses.

- void **Flip**(Literal p)
  This flips the truth value of the proposition in the literal and updates the solvers internal data structures (see below).

- int **SatisfiedClauseDelta**(Proposition p)
  Returns the increase or decrease in the total number of satisfied clauses if we were to flip the specified proposition.

We've written the rest of the code for you. **Do not modify any other files**, or we may not be able to grade your assignment.

Data structures

The solver maintains the following data structures:

- **Problem**: represents a SAT problem and implements WalkSAT.
- **Proposition**: a proposition in a problem. A proposition has a name (a string) and lists of constraints it appears in.
- **Literal**: a proposition and whether the proposition is negated.
- **Constraint**: basically an array of literals, along with a minimum and maximum number of literals that can be true in order for the constraint to be satisfied.
- **TruthAssignment**: an object representing a truth assignment. It acts like an array, so if a is a truth assignment and p is a proposition, then a[p] is p's truth value in a.

Begin by reading the code in Proposition.cs, Literal.cs, Constraint.cs and TruthAssignment.cs. That will show you what the basic representation of clauses is and what methods are available for working with them.

Bookkeeping inside the solver

The solver spends a lot of time:

- Looking for unsatisfied constraints (constraints that are false given the current truth assignment)
- Checking how many literals of a constraint are satisfied (true in the current truth assignment)

It does these enough that they dominate its execution time. We could do these by, for example, constantly iterating through all the literals of a constraint, checking their individual truth values, and adding them up. But that's unacceptably slow.

Instead, solvers maintain tables of the number of satisfied literals in each constraint (this is the array TrueLiteralCounts[] in Problem.cs). Each constraint is numbered, and the number is in its field called Index. So if c is a constraint, the number of true literals currently in it is TrueLiteralCount[c.Index]. You can get at this easily just by calling CurrentTrueLiterals(c).

A constraint c is unsatisfied if CurrentTrueLiterals(c) less than the minimum number of literals or more than the maximum. You can check this with Satisfied(c) and Unsatisfied(c). However,

we need to be able to find unsatisfied constraints quickly without having to iterate through all the constraints every time. So the solver also maintains a table of unsatisfied constraints called, unsurprisingly `UnsatisfiedConstraints`. It's of type `List<Constraint>`, which in C# behaves like an array except you can add things using `.Add()` and remove them using `.Remove()`. So it's like an ArrayList in Java or a Vector in C++.

Here's the hard part

We wrote the code to initialize those data structures. But when you write the `Flip()` method, you need to make sure that it not only flips the value of the variable, it also updates the `TrueLiteralCounts` and `UnsatisfiedConstraints`, since `StepOne()` will depend on them being accurate to select constraints and literals within the constraints. If they're inaccurate, everything falls apart in ways that will be very difficult to debug.

Debugging tools

The good news is that we also wrote a method ConsistentCheck() that makes sure the tables are correct, and the GUI code calls it after every step to make sure nothing is amiss. If it fails, an exception will appear in the Console window at the bottom of the Unity editor.

We've also given you a GUI (demoed in class) you can use to single-step the solver and display all its state on the screen. You can also do code-level single stepping by pressing the Attach to Unity button in Visual Studio, Rider, or MonoDevelop, whichever code editor you had Unity install.

Finally, Unity provides a the call Debug.Log(string message) to dump things to the Console window in Unity.

What to do

1. **Download and install** the current version of Unity.
   You can download it here.

2. Now **read all the code in the SAT directory**.
   There really isn't very much of it. You aren't trying to memorize it or anything. Just familiarize yourself with what the public methods and fields of all the classes are. And familiarize yourself with the private fields of the `Problem` class. **Make sure you learn where it is that the `Problem` class stores the current truth assignment it's working on**. You will need to know that for the next part, which is to…

3. **Write `Flip()`**.
   Your implementation can call the Flip() method of the truth assignment, to actually change the truth value. But you need to **make sure you correctly update the `TrueLiteralCounts` and `UnsatisfiedConstraints` tables**. To do that, you'll want to look at the `PositiveConstraints` and `NegativeConstraints` fields of the proposition you're flipping. These give all the constraints the proposition appears in either in positive (unnegated) form, or in negative (negated) form. If you flip a proposition from false to true, then all its positive constraints have their true literal counts go up and all its negative constraints have their counts

go down. And it's the other way around, which the proposition flips from true to false. When a constraint's TrueLiteralCount go outside the range from its MinTrueLiterals to its MaxTrue literals, it need to be added to UnsatisfiedConstraints. And when it rturns to that range, it needs to be removed from it.

Write this now, but it will be easier to test after you do step 4, below, which will be easy. It will call `Flip()` for you and you can try it out by pressing the **Step** button in the GUI, which will also consistency check your tables.

4. Write a simplified version of **StepOne()** that implements just the random walk solver from slide 15 of lecture 11. This just picks a random clause, and then picks a random literal from it, and flips that. We've added a `RandomElement()` method to the array and list classes so you can just say things like `UnsatisfiedClauses.RandomElement()` and not have to mess with making a random number generator.

    Having done this, try testing it out and getting the code to work. It should be able to solve problems.

5. Now **implement SatisfiedClauseDelta(Proposition p)**
    This tells you whether flipping p would increase or decrease the total number of satisfied propositions.

6. Finally, implement the **full WalkSAT algorithm**
    This is found on slide 21 of lecture 11, but also reread slide 17. This will use `SatisfiedClauseDelta()`. And you'll find the `Random.Percent()` method useful too to choose between the two strategies based on the `NoiseLevel`.

Congratulations! You've implemented the important parts of a SAT solver.

Turning it in

To turn it in, **upload just your Problem.cs file** to canvas.