Lab 8

Due: Tuesday 10/31/2023 @ 11:59pm EST

The purpose of labs is to practice the concepts that we learn in class. To that end you will be writing java code that uses a game engine called Sepia to develop agents that solve specific problems. In this lab we will be playing a traditional single-player game Minesweeper. In order to make Minesweeper work in Sepia, I had to change it to a two-player game. Fear not, the rules of Minesweeper will remain the same: except now there is a (reactive) enemy agent controlling the board (to replace units when you attack them, etc).

1. Copy Files

Please, copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/lab8/lib/minesweeper.jar` to `cs440/lib/minesweeper.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/lab8/data/lab8` to `cs440/data/lab8`.
  This directory contains a game configuration and map files.
- Copy `Downloads/lab8/src` to `cs444/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/lab8/lab8.srcs` to `cs440/lab8.srcs`.
  This file contains the paths to the `.java` files we are working with in this lab. Just like last lab, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- `Downloads/lab8/doc`. While you don't have to copy this anywhere, this is the documentation generated from `minesweeper.jar` and will be extremely useful in this assignment.

2. Test run

If your setup is correct, you should be able to compile and execute the given template code. You should see the Sepia window appear.

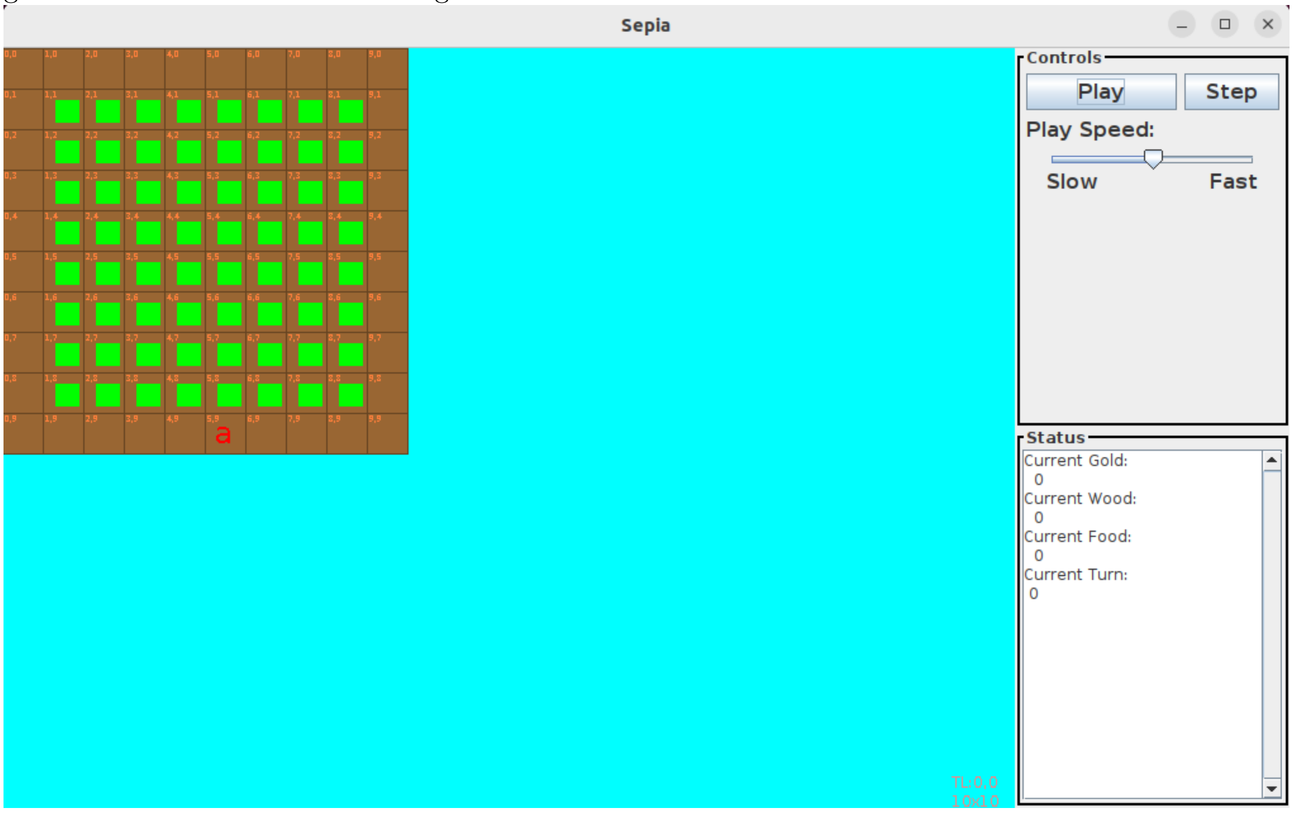
```
# Mac, Linux. Run from the cs440 directory.
javac -cp lib/Sepia.jar:lib/minesweeper.jar:. @lab8.srcs
java -cp lib/Sepia.jar:lib/minesweeper.jar:. edu.cwru.sepia.Main2 data/lab8/easy/8x8.xml

# Windows. Run from the cs440 directory.
javac -cp lib/Sepia.jar;lib/minesweeper.jar;. @lab8.srcs
java -cp lib/Sepia.jar;lib/minesweeper.jar;. edu.cwru.sepia.Main2 data/lab8/easy/8x8.xml
```

3. Minesweeper Rules & Information

The game of Minesweeper is simple. You are given a grid of squares, the contents of each square are initially hidden from you. The player interacts with the game by clicking on a (currently hidden) square, at which point the contents of that square are revealed. A square can contain two kinds of things: a mine, or nothing. If you click on a square that reveals a mine, the mine blows up and you lose. If you click on a sqaure that does not have a mine, you are instead told how many mines exist in adjacent squares. In most implementations of Minesweeper, that number of adjacent mines is drawn inside the square except for if there are no adjacent mines. In the case of no adjacent mines, Minesweeper implementations typically reveal all non-mine squares around the empty square (for human convenience). The first square to be clicked is always safe (the game enforces this by moving mines around if you happen to click on a mine initially), and the game ends when all non-mine squares have been revealed (i.e. clicked on).

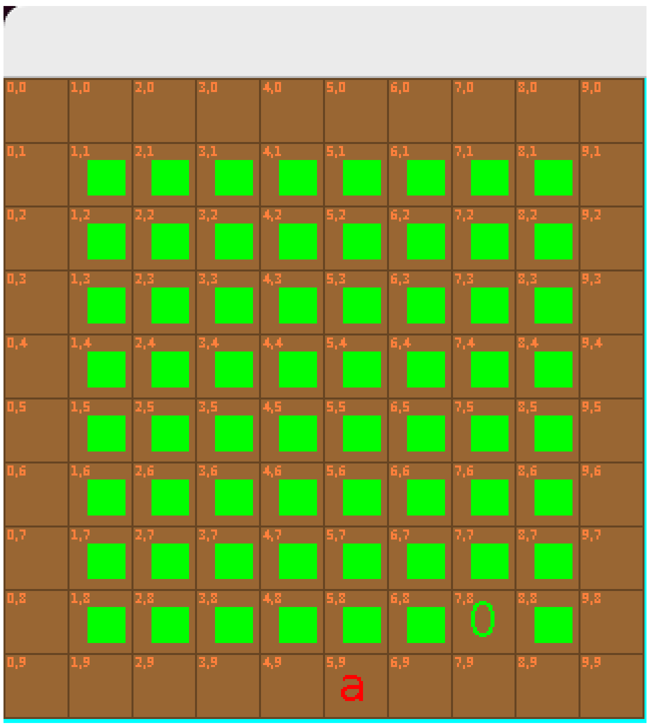
Our version of Minesweeper is similar. Let's say that you run `data/lab8/easy/8x8.xml`. You should get a window that looks something like this:



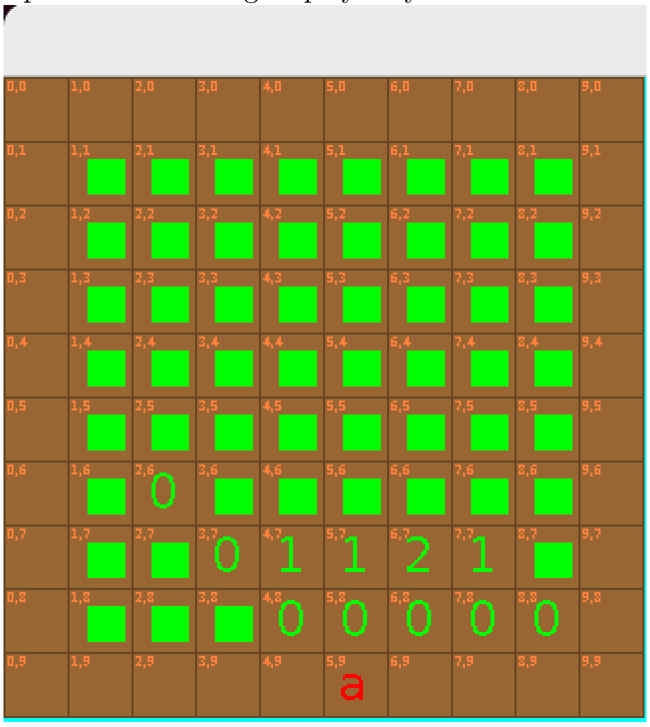
```
Current Gold:
 0
Current Wood:
 0
Current Food:
 0
Current Turn:
 0
```

Any square that has a green box within it are hidden (i.e. unclicked on) squares. Your agent controls the red "a" at the bottom of the grid. The border of unmarked squares are not part of the game: they are only there so that the "a" has somewhere to exist on the map.

Your agent will play Minesweeper by attacking squares in the map. Your agent is only allowed to attack hidden squares (i.e. squares with green boxes in them): attacking any other square will result in that square firing back at you (and killing you). Your unit as well as all hidden squares die in one hit. Once a hidden square has been killed, the enemy agent will replace that unit with a new unit whose ascii-art shows the number of mines adjacent to that square. For instance, consider your agent attacking square (7,8) and seeing the following change:



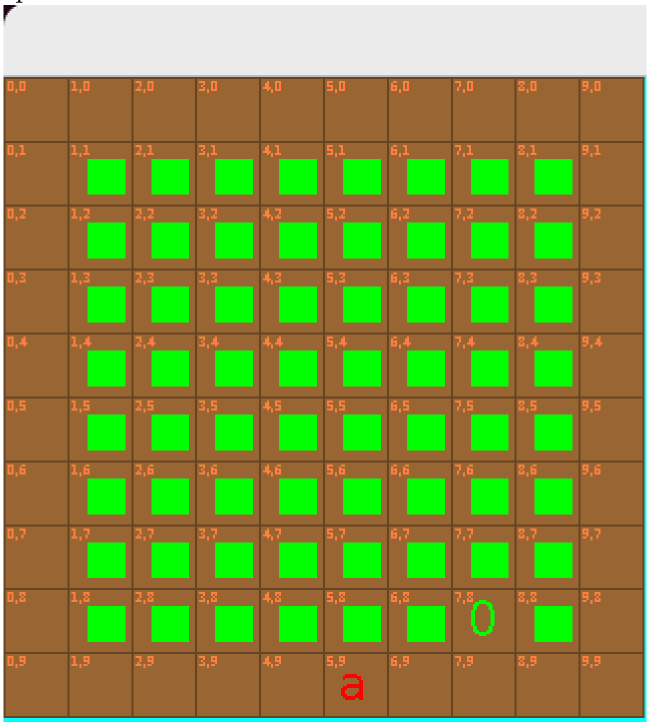
The new unit that appears in square (7,8) shows that there are zero mines hidden in the adjacent squares. Continuing to play may result in a board that looks like:



Here we can see that there are two mines adjacent to (6,7), and one mine adjacent to (4,7) as well as (5,7) and (7,7). Every time it is your agent's turn to play, your agent must pick a square to attack. If a mine appears, that square will contain an asterisk *. Any mine revealed will attack you next turn (and kill you).

4. Sentences & Inference Rules

Your agent needs to use the number of adjacent mines (called "clues") to figure out which squares are safe, and which squares contain mines. We will be doing this via logic (i.e. constructing sentences and using a knowledge base to infer new sentences). The good news is that our sentences have a very specific format. Consider the world shown earlier:



When we observe a (non-mine) square, we are told how many mines are adjacent to that square. So, we will construct a variable per square (i.e. coordinate), and sentences will relate variables to the number of mines contained within them. If a square with coordinates c reveals number n, then we know there are exactly n mines adjacent to coordinate c. We can construct the following sentence:

$$\{c' | adjacent(c, c')\} = n$$

For example. In the board above, the coordinates adjacent to (7,8) are $\{(6,8),(6,7),(7,7),(8,7),(8,8)\}$, so the sentence we would construct for observing 0 in (7,8) is:

$$\{(6,8),(6,7),(7,7),(8,7),(8,8)\} = 0$$

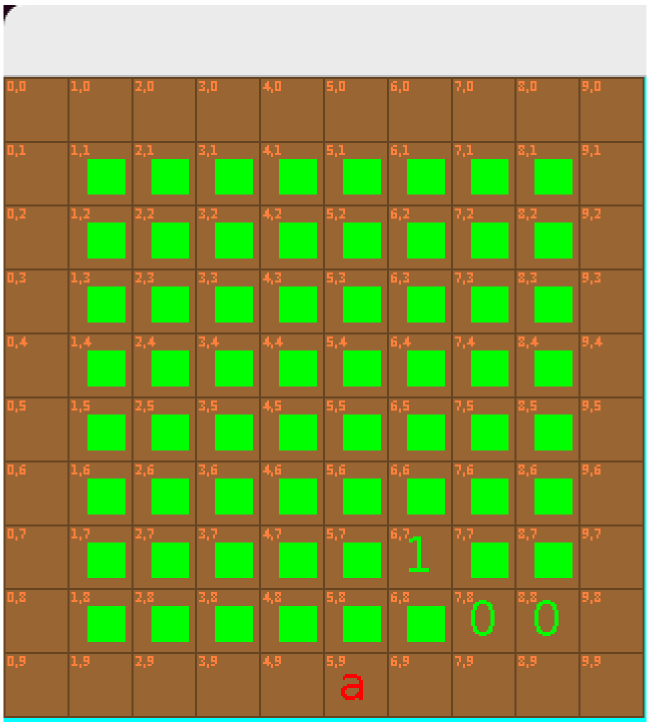
where $\{(6,8),(6,7),(7,7),(8,7),(8,8)\}$ are called the *coordinates* of sentence s and 0 is the *constraint* of s. Of course, given the above sentence, we should infer that all of the coordinates of s are safe. This leads us to our first inference rule: **If sentence s has constraint 0, then all of the coordinates of s are safe.**

Likewise, let us consider where we click on the corner and a 3 appears. Let's pretend the corner we click on has coordinates (1,1). The sentence we would construct is as follows:

$$\{(2,1),(1,2),(2,2)\} = 3$$

Since there are three coordinates in s and the constraint of s is also three, we should be able to infer each of those coordinates contains a mine (i.e. is *not* safe). This leads us to our second inference rule: **If the cardinality of sentence s is equal to the number of coordinates in s, then all of the coordinates of s are mines.**

Finally, let us consider the following scenario:



When we attack coordinates (6,7) and see the 1, we could construct the following sentence:

$$\{(5,7),(5,8),(6,8),(7,8),(7,7),(7,6),(6,6),(5,6)\} = 1$$

However, we know that (7,8) is safe (because it contains a zero). In fact we should have also inferred that (6,8), (7,7), and (8,7) are safe as well (due to them being adjacent to (7,8)). So, we should be able to *simplify* the above sentence to:

$$\{(5,7),(5,8),(7,6),(6,6),(5,6)\} = 1$$

While we may not be able to fully solve this sentence now, we made it much simpler. This leads us to our third inference rule: **If we know coordinate c is safe, then we can remove c from any sentence that contains c (without changing the constraint of c). If we know c to be a mine, then we can remove c from any sentence that contains c and also decrement the constraint of c by one.**

Task 1: Knowledge Base Inference (100 points)

In this task, I want you to implement the three inference rules mentioned in the previous section. I have left you quite a bit of starter code, so to be clear the three inference rules should be implemented in the following methods:

- `src.lab8.agents.LogicAgent.KnowledgeBase.inferAllSafeCoordinates`: This method is where you should implement the first of our inference rules: that if a sentence has constraint 0, then all of the coordinates inside that sentence are safe (and can be attacked).
- `src.lab8.agents.LogicAgent.KnowledgeBase.inferAllMineCoordinates`: This method is where you should implement the second of our inference rules: that if a sentence has constraint n and also has n coordinates inside it, then every coordinate inside that sentence can be infered to be a mine.
- `src.lab8.agents.LogicAgent.KnowledgeBase.simplifySentences`: This method is where you should implement the third of our inference rules: that if a sentence contains a coordinate that is known to be safe, we can remove that coordinate from that sentence. Likewise, if a

sentence contains a coordinate that is known to be a mine, we can remove that coordinate from that sentence and also decrease the constraint of that sentence by one.

Notes

- Whenever you want to test your implementation, I would recommend playing any of the three games in the `data/lab8/easy` directory. Like the name of the directory states, there are easy games, medium (difficulty) games, and hard games. I would recommend testing your code initially on the easy games and then moving on to trying the medium and hard games. Here are the games that you can run for each difficulty:
  - `easy`: There are easy games with three different sized boards: 8x8, 9x9, and 10x10. They are contained in the `8x8.xml`, `9x9.xml`, and `10x10.xml` files. Each easy game has 10 hidden mines.
  - `medium`: There are two medium games with different sized boards: 13x15 and 16x16. They are contained in the `13x15.xml` and `16x16.xml` files. Each medium game has 40 hidden mines.
  - `hard`: There is only one hard game: a 30x16 contained in the `30x16.xml` file. This game has 99 hidden mines.
- I would recommend implementing `inferAllSafeCoordinates` first. This rule alone is very powerful: powerful enough to let you win easy games better than chance.
- I would then recommend implementing `inferAllMineCoordinates`. While it may not have an obvious impact on your agent's performance (without `simplifySentences`), this rule is delicate. Whenever we find a sentence that this rule applies, we want to (after marking each coordinate as a mine), evict this sentence from our knowledge base. Therefore, pay special attention not to get `ConcurrentModificationExceptions` as you delete from the KnowledgeBase's data structure while you traverse them. What I chose to do was to rebuilt the KnowledgeBase's data structures with only the sentences that this rule does *not* apply to. This will avoid deleting and traversing the same data structure.
- Lastly I would recommend implementing `simplifySentences`. This method, just like the last one, can involve deleting while traversing. So, I implemented this also to rebuild the data structure's inside the KnowledgeBase to convert deletion into addition. This is the hardest of the three to implement, but is also the core of the inference procedure. When implemented correctly, your agent should be able to solve easy and medium difficulty games pretty consistently.
- You may create whatever helper methods you want in order to accomplish this goal, however I am not sure that you will need to create any in `LogicAgent.java`.

Task 2: Extra Credit (50 points)

There is a fourth inference rule that we can implement if we're being clever. It involves merging two sentences s_1 and s_2 together to produce a new sentence s_3. Since this is extra credit, I will not be giving you the logic for what kind of sentences we can merge, nor how to merge them. That is up to you!

Please create and implement `src.lab8.agents.LogicAgent.KnowledgeBase.mergeSentences`. This method will calculate all merged sentences (it is a good idea to discard sentences once you are sure you're done with them). Remember to add your method call to `src.lab8.agents.LogicAgent.KnowledgeBase.makeInferences` so your changes can take effect! Once this method is implemented correctly, you should be able to solve the hard game pretty consistently.

Task 3: Submitting your lab

Please submit `LogicAgent.java` on gradescope (just drag and drop in the file). If you tried the extra credit, the autograder will use inflection to detect whether or not method `src.lab8.agents.LogicAgent.KnowledgeBase.mergeSentences` exists: and it will play a bunch of hard games (and count the number of successes). Be warned, this can take a while.