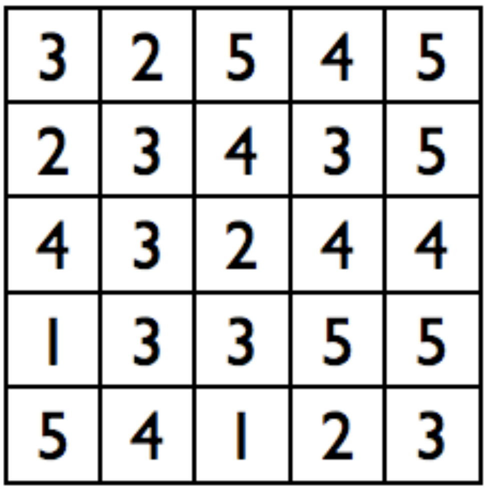
AI Project #1: Hitori

Due Monday 1/7/13, 11:59 PM

Hitori is a puzzle type invented by Nikoli. It is not my favorite Nikoli puzzle (though that is sort of like being my not-most-favorite flavor of Ben and Jerry's). So let's create a program that solves them for me.

A Hitori puzzle initially looks like a square grid of numbers (1-n for an n by n grid), something like this example:

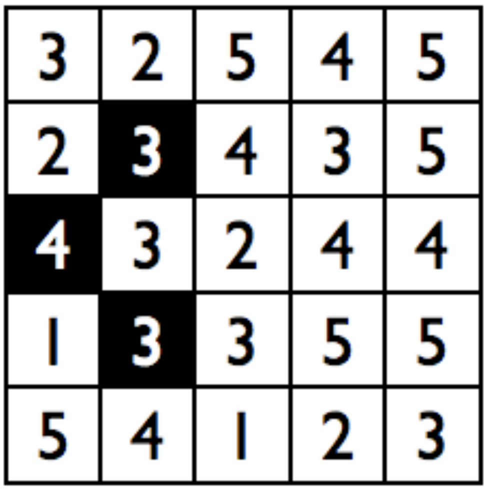


3	2	5	4	5
2	3	4	3	5
4	3	2	4	4
1	3	3	5	5
5	4	1	2	3

The goal in a Hitori puzzle is to color some of the squares black (and leave the rest white) such that:

1. When you are done, the same number does not appear twice (unblackened) in any single row or column.
2. No two black squares are horizontally or vertically adjacent (diagonal is OK).
3. All of the white squares form a single (horizontally and/or vertically) connected group.

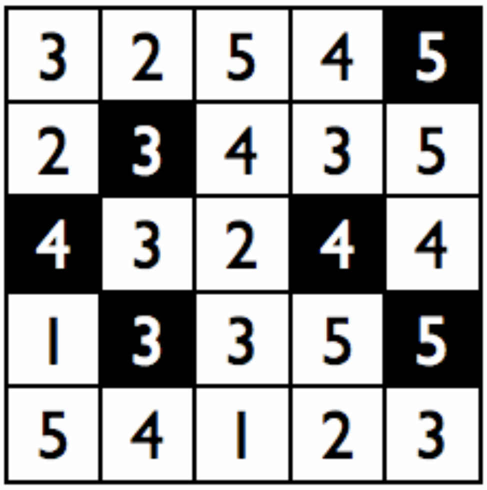
To see how these rules might be used by a human to solve a puzzle, let's look at the above example (we will discuss below how a computer's solving process will differ). In the above example, consider the three 3s in the second column. Due to rule 1, at most one of them can remain white. If we blacken the middle one, we cannot blacken either of the other two due to rule 2. So we leave the middle one white and blacken the other two. Next consider the three 4s in the middle row. In this case, if we leave the one on the left white, the other two would have to be black, violating rule 2. So it must be black. That gives us:



3	2	5	4	5
2	■3	4	3	5
■4	3	2	4	4
1	■3	3	5	5
5	4	1	2	3

(Note that this image does not differentiate "Must be white" from "Not sure yet". This will be a very important distinction in your solving process.)

Finally, we can invoke rule 3 to determine that the 2 in the middle cannot be black, because this would isolate the white 3 next to it. Likewise neither the 5 and the 1 in the middle column can be black. Making the 5 white causes the 5 in the upper right to be black, and so on until we reach the solution (in which you can verify that all rules are satisfied):



Because this is a properly constructed Hitori with only one solution, at the end all squares are known to be either white or black - that is, changing the assignment of any square would violate a rule of the puzzle.

So now, on to the solver:

To complete this project, you will write a constraint satisfaction solver in Lisp (`clisp`), Scheme (`racket`) or Python (`python3`) that solves Hitori puzzles. You should provide a function `solve-hitori` that expects a reasonable representation of the initial puzzle grid and produces either a reasonable looking solved puzzle or "no solution" if there is not one (if a puzzle has more than one solution, your solver only needs to give one).

For the purposes of this assignment, you will implement both a brute-force version of the solver as well as a more intelligent one. As such, your `solve-hitori` function should take an additional argument indicating which solving method should be used. For the brute-force version, you should simply pick the next square to assign based on some fixed order, rather than using Minimum Remaining Values, however you should still eliminate obviously invalid configurations right away. For the intelligent version, you do not need to represent all of the constraints explicitly in your program, but you must use forward checking of both rules 1 and 2 (not rule 3 unless you really want to). That is, turning a cell white should cause checks and elimination of a possible value for other cells due to rule 1, and turning a cell black similarly due to rule 2. Along with (and perhaps immediately as a result of) these forward checks, the Minimum Remaining Values heuristic can then be applied, since you have necessarily reduced the possible values from 2 to 1 for all relevant cells. However, once you have no cells that are forced to a particular color (such as at the beginning of the search process), you may choose which cell to try assigning a value to using any method you choose. For both versions of the solver, you should also keep track of (and print out!) the total number of states generated, to see the effects of the intelligent search versus brute force.

For your initial testing, you may wish to use smaller puzzles for easier tracing of your code. Here are two small Hitori with unique solutions, and one with no solution (this would be considered "broken" as a puzzle, but is helpful for testing purposes):

```
2 1            1 2 3          1 2 3
1 1            1 1 3          2 2 3
               2 3 3          1 1 3
```

For efficiency testing, you will probably want to use larger puzzles, such as at the Nikoli page. As a point of reference, my solution (in Lisp) solves puzzle 4 on that page in under a second using forward checking (generating <400 states counting each additional square coloring as a new state) and using brute-force takes 3-4 seconds and >3000 states. You will likely get different numbers depending on your implementation but I would expect similar levels of efficiency.

Submission

Once you are satisfied, you should submit your code along with documentation via any CS Unix machine using the command

```
submit zjb-grd proj1 file-name(s)
```

Your documentation must explain and defend your state representation and describe how your solver operates. You should also, as an example, provide a function call that solves the example puzzle given at the top of this page.

Note that the `submit` command gives no feedback in case of a **successful** submission!

Grading

I will grade your submissions as follows:

- Finds correct solutions to puzzles: 40 points
- Performs forward checking and MRV as specified: 40 points
- Quality and defense of state representation: 10 points
- Cleanliness/efficiency of code and execution: 10 points