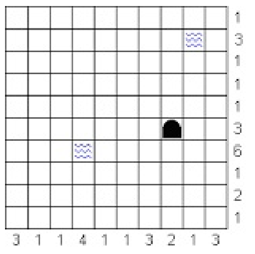
Assignment 3: CSP Battleship

Document History

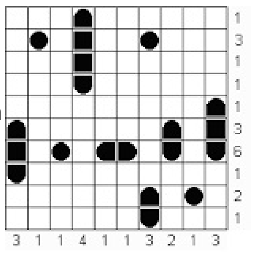
- Randy added some tips in the Suggestions section Oct 27 2022
- **Validation scripts (https://q.utoronto.ca/courses/278996/files/22891311?wrap=1) (https://q.utoronto.ca/courses/278996/files/22891311/download?download_frd=1)** provided Oct 25 2022
- Fixed some tyops Oct 20 2022
- Released Oct 19 2022

Overview



In this assignment you will be creating a CSP solver for the domain of Battleship Solitaire puzzles. This will require you to encode these puzzles as a constraint satisfaction problem (CSP), implement the CSP solver, and use that to solve the puzzles we provide.

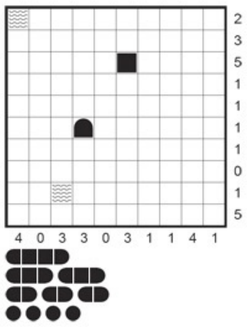
Background



Battleship Solitaire (which also goes by other names) is a game played on a square NxN grid that contains ships of different sizes and sections of open water, similar to the **Battleship board game (https://en.wikipedia.org/wiki/Battleship_(game))**. Unlike the board game (which is meant to be played against another player), Battleship Solitaire shows the number of occupied squares in each row and column, from which you're meant to determine the location of each ship (see Figure 1, at right).

To aid with this, these puzzles observe the following rules:

1. There are four types of ships in these puzzles, each of which occupies a specific number of squares:
   - Submarines (1x1)
   - Destroyers (1x2)
   - Cruisers (1x3)
   - Battleships (1x4)
2. Ships can be oriented vertically or horizontally, but not diagonally. For example, the battleship can either be a 1x4 rectangle or a 4x1 rectangle, depending on the orientation.
3. The number of ships for each type is provided as part of the puzzle (see Figure 2, at right).
4. Ships are not allowed to touch each other, even diagonally. This means that each ship must be surrounded by at least one square of water on all sides and corners.
5. In addition to being provided with the number of occupied squares in each column and row, some puzzles also reveal the contents of certain squares, showing whether they contain water or some piece of a ship. Where a ship piece is revealed, it will indicate whether it is a middle or end portion of a ship, and in the case of the end portion it will show what the orientation of that piece is (i.e. what direction the rest of the ship can be found).
   - In the case where a submarine is revealed, it will simply show the entire ship.



For more information or for the rules of battleship solitaire, please consult **https://en.wikipedia.org/wiki/Battleship_(puzzle) (https://en.wikipedia.org/wiki/Battleship_(puzzle))**. You can play games of battleship solitaire for free at **https://lukerissacher.com/battleships (https://lukerissacher.com/battleships)**.

What you need to do

In order to solve these puzzles, you need to choose variables for this problem and define a domain for each variable. You then need to define enough constraints over the variables that would sufficiently capture the rules of Battleship Solitaire.

Remember to choose constraints wisely such that when solving you will be able to prune domains of variables efficiently and shrink the search space. There are a few tips we recommend to accomplish this:

- Design your variables in a way that makes it easy to check the constraints.
- Avoid variables that require an exponential number of values.
  - Performing GAC on such constraints will be too expensive.
- Try not to use table constraints over large numbers of variables.
  - Table constraints over two or three variables are fine: performing GAC on table constraints with large numbers of variables becomes very expensive.

You then need to implement a CSP solver. The implementation details are up to you, but it would be advisable to follow the concepts used in this course. Try using forward checking and general arc consistency, along with a backtracking search, and see which works better.

Evaluation

Your submission will be marked primarily for correctness and performance. We will test your solution on grids that range from 6x6 to 10x10 with a unique solution. Your solver needs to find this unique solution (correctness) within the time allotted (~5 minutes per puzzle). There is a small portion of your assignment mark allocated for the explanation of any heuristics that you used.

Input Format

Your program should be named `battle.py` and take in two arguments from the command line: the names of the input and output files. The following command should run your program on the puzzle in `puzzle1.txt`, storing the output in `output1.txt`:

```
python3 battle.py puzzle1.txt output1.txt
```

The input file will represent an NxN puzzle and be formatted as follows:

- On the first line will be the puzzle's **row constraints** written as a string of N numbers (note that row constraints are usually written to the left or the right of each row when viewing examples of these puzzle).
- On the second line will be the puzzle's **column constraints** written as a string of N numbers (note that column constraints are usually written on top or bottom of each column when viewing examples of these puzzle).
- On the third line will be 1-4 numbers, representing the number of submarines, destroyers, cruisers and battleships, in that order. Some of the smaller puzzles are too small to contain larger ships, so if there are less than 4 numbers in this row you can assume that no ships of that size exist in this puzzle.

The remaining lines will represent the starting layout of the puzzle, which indicates any starting values you have to work with. Each line will represent a row of the Battleship Solitaire grid as a string of characters, with each character representing a single square in that row. There are 8 possible characters for this section of the input file:

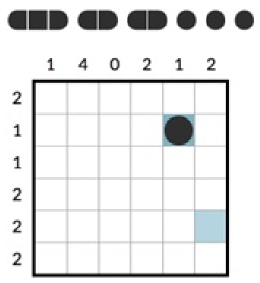
- '`0`' (zero) represents no hint for that square
- '`S`' represents a submarine,
- '`W`' represents water
- '`L`' represents the left end of a horizontal ship,
- '`R`' represents the right end of a horizontal ship,
- '`T`' represents the top end of a vertical ship,
- '`B`' represents the bottom end of a vertical ship, and
- '`M`' represents a middle segment of a ship (horizontal or vertical).

Therefore, an NxN puzzle will have its initial grid represented by N lines, each of which is N characters long.

An example of an input file would be:

```
211222
140212
321
000000
0000S0
000000
000000
00000W
000000
```

The above input file corresponds to the puzzle below (in typical print form).



As stated above, your main program should take two command-line arguments: one for the input file and one for the output file:

```
    python3 battle.py <input file> <output file>
```

For example, the command `python3 battle.py input1.txt output1.txt` will take in the input file as described above, stored in `input1.txt`. Your submission would store the solved grid in file `output1.txt`. The input and output files will both be in standard text format.

Output format

The output will be similar in format to the input, except only the contents of the final solution to the NxN board will be printed. Each character in the output corresponds to the contents of a cell in the solution. Similar to the input file, there are 7 possible values for each square in the solution grid (there should be no '`0`' characters left once the puzzle is solved). For example, the correct output for the input example above would be:

```
LRWWWW
WWWWSW
WTWWWW
WMWWWS
WBWTWW
WWWBWS
```

Validation Script

A **validation script (https://q.utoronto.ca/courses/278996/files/22891311?wrap=1) (https://q.utoronto.ca/courses/278996/files/22891311/download?download_frd=1)** has been provided to help you check that you're satisfying the input/output requirements, along with several test puzzles. As a reminder, these will be a subset of the cases that we'll use to test your submission for correctness.

Time and Space Constraints

Your AI agent has at most 5 minutes to make each move on the teach.cs server. While this might be sufficient time to solve the smaller grid with forward checking alone, the larger grids will need GAC coupled with some heuristics of your own. We may raise the timeout limit for some of the larger grids, but only by a few minutes. As always, your program should be able to run without causing any memory exceptions on the teach.cs servers under typical conditions (we typically test your submission when there is a light load of jobs running on the teach.cs server).

Suggestions

- If you need to make your solution more efficient to meet the timeout constraints, consider implementing variable selection heuristics or constraint ordering heuristics (such as MRV).
- Preprocessing a little bit can help (especially if your FC/GAC implementation is inefficient). For example, you sometimes know you can pad certain squares with water before search even begins and you know columns/rows that add up to 0 are all water.
- Split as many n-ary constraints into binary constraints as possible. This will propagate values faster. This is absolutely critical if you are using FC, but even helpful if you are doing GAC. Similarly, constraints with fewer variables are preferable to constraints with more variables (for faster propagation).
    - That being said, if you have some non-binary constraints, consider doing GAC instead of FC. Never do plain backtracking.
- Follow the GAC psuedocode from the slides or textbook and pay attention to the data structures you use. There are many ways to implement GAC and some are not efficient (e.g., look up AC-1 vs AC-3). There is also an AC-4 which is generally better, but you'll have to look outside the textbook for it.
- You could try encoding ships as variables, but a warning this is not an easy way to model the problem. It might reduce your search complexity if done correctly though.
- Consider creating a lookup table for past GAC supports, to save you time when revisiting variables in the GAC enforce algorithm.
- There are binary constraints that might help propagation. For example, every square occupied by anything other than W or 0 must have each of its diagonals be W. So every pair of diagonal squares (x, y) on the board can have a constraint C(x, y): x == 'W' or y == 'W'. Doing FC or GAC on an OR constraint should be easy to figure out.
- More suggestions to come as we think of them :)

Mark Breakdown

As stated earlier, the majority of your marks will come from correctness and performance, in that we measure how many puzzles you are able to solve within the time provided.

Marking Scheme

Component	Weight
Correctness (puzzles solved)	90%
Heuristic file	10%

As shown in this table, 10% of your mark for this assignment is earned through the implementation of additional heuristics or techniques that you add to your CSP solver. These are meant to be part of your solution and described in a file called `heuristics.pdf`, which you will submit on Markus along with your `battle.py` file. This mark is assigned separately from your correctness mark (i.e. you can get full marks for your heuristics even if you don't get full correctness marks).