Younjee Kang
1005978621

CSC384 - A3: Battleship Solitaire CSP Heuristics

Modelling:

We have tried different models and found the design that has each ship being variable to be better. For instance, using cell-based models has the advantage that the domain size is a lot smaller than the ship-based model. More precisely, the cell model I planned to use had a domain of at most 5 (W, S, L, M, B, R, T), whereas the initial ship model had at most 100 values(10x10) in each domain. However, the cell model has at most 100 variables, the cell-based model still needs a very expensive, rigorous search. Plus, as we can reduce the domains of each ship (variables), we found this ship-based model not to be so expensive, yet more optimal if well designed.

Preprocessing:

Making sure we do not want to search with obviously-failing values一 *i.e., what are the values that we humans can easily infer from the start?* 一, we have worked to reduce the domain of each variable as much as possible.

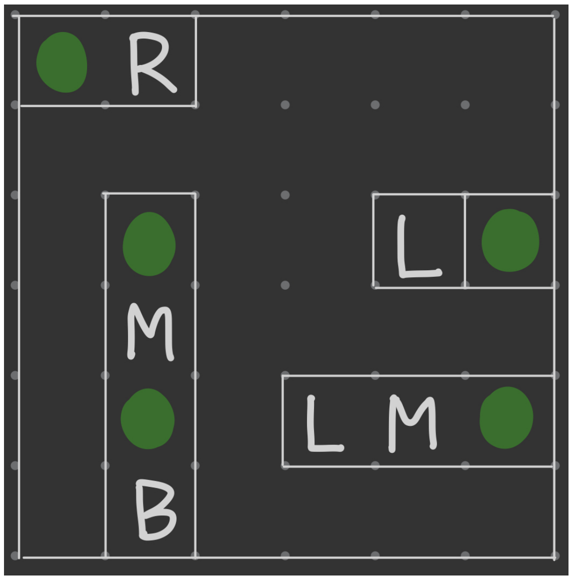
First, we have constructed a lot of detailed constraints. Then, we realized that some of the constraints could be used in the preprocessing step so that the algorithm can easily start with much reduced domains. Doing this reduces the time of checking if the assignment falsifies the constraint and also the time even for assigning the value in the first place. This includes:

1. Every ship should be a **whole ship**.
    a. **Heads**(T, L) must be followed by **tails**(B, R) after at most 2 M pieces, depending on the ship kind.
    b. Every piece of it should be **on the grid**, so for each kind of ship, for each orientation(whether horizontal or vertical) the possible starting points are restricted. (i.e., the heads of the horizontal long ships cannot be placed on the last column.)

2. **Preassigned** Pieces
    a. Submarines cannot be on any of the preassigned W, L, M, R, T, B or near the L, M, R, T, B.
    b. Destroyers cannot be on any of the preassigned W, M, or S pieces or any near M, S.
    c. Cruisers cannot be on any of the preassigned W, S and any near S pieces.
    d. Battleships cannot be on any of the preassigned W, S, and any near S pieces.
    e. The hinted squares, especially if it is a ship piece, must be occupied by exactly one ship.
        i. For example, for a piece S, one submarine can start off with only one value in its domain, and other ships can get rid of the value from their domains. This is possible as an S piece has only one possible kind of ship that can occupy it.
        ii. Not only S, if there is any preassigned piece that must be in a certain kind of ship, then a ship of that kind must be placed there. I.e., a case where

Younjee Kang
1005978621

humans can easily guess the remaining piece other than what is revealed. Some examples of this case are illustrated below:



← The green pieces can be easily guessed. In other words, we can simply assign them to one of the corresponding ship kinds from the start.

Caching Explored Set:

Using sets, we cached the value(positions that the current variable is assigned). Note we do not need to cache the variable together. The position tuple contains exactly the same number of square coordinates if variables share their kinds, and this is sufficient. This is because the solution does not care which submarine it is, but rather considers all the ships the same if they have the same kind.

Plus, if it failed to put a submarine in some place by some constraints with current assignments, it means placing another submarine in that place will fail with any extended assignments. In other words, we should avoid assigning a variable to a position if another ship was occupied, even before being checked in one of all the constraints.

MRV

When picking an unassigned variable, the algorithm picks a variable with **minimum remaining values**. This is because if a variable has only a few choices left in the first place, and if any of that small set falsifies constraints, it doesn't need to explore further since there is no solution in them eventually.

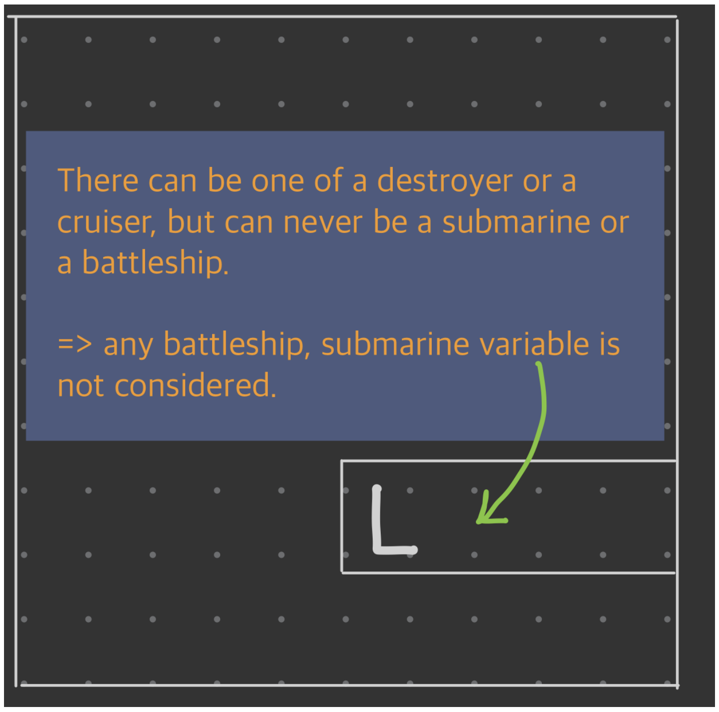
Node Ordering Heuristic

Other than MRV, we also took into account the **length of the ships**. If we have unassigned longer ships, we pick that ship to consider first. **The longer the ship is, the more likely it will**

falsify constraints, such as row and column constraints, touching(=colliding) constraints, and so on, because it occupies more squares.

Constraints Scope Reducement

When we have a higher-order constraint, the complexity is very high as there is a very large domain for each variable in our model. So it is crucial to reduce the scope of every constraint as much as possible. We attempted to achieve this in plenty of ways; One example is, as shown in the diagram, when checking whether every preassigned ship piece is occupied by some ship(i.e., a constraint that checks if a particular ship piece is occupied.), we can reduce the number of variables as much as possible, rather than exploring the values of all the ships(variables). Here, as we know it can only be either destroyer or a cruiser, only variables of that kind would be considered.


There can be one of a destroyer or a cruiser, but can never be a submarine or a battleship.
=> any battleship, submarine variable is not considered.
L

Another example would be in a **row-column** constraint, a.k.a. a table constraint, we first had a constraint that adds the total number of occupied squares in each row and column, and see if it is equal to the corresponding value. Doing this was very expensive because, for each row and column, we have to look at all the variable-value pairs to get the total. Even though we had an idea that if the total is greater than the value in any point of addition, it is falsified immediately, it was not enough. Then, we found that we only need to care about the remaining row and column constraint as the puzzle is meant to be solved in a perfect match of row-column values, correctly occupied preassigned pieces, and the total number of each ship on the grid. In other words, we do not need to care about all variables and wait till all variables get assigned by some value, everytime we assign a new value, we can check if this value does not falsify the remaining row-column constraint(that is <0). To do this, we ended up carrying the row and column dictionary and subtracting the corresponding value when a compatible variable was assigned. But this way, we ended up having a **unary constraint** from a **higher order constraint**, specifically, the one that cares all variables.