Battleship

Project Type: ***CSP + Search***

Member	**teachcsID**	**Student No.**	**Role in Project**
Iven Poon	*pooniven*	*999169607*	*Tester (major), Evaluation, Evaluation, Report*
Shih-ying Jeng	*jengshih*	*1003707884*	*Model (major), Evaluation, Report*
Peiyu Liao	*liaopeiy*	*1003707893*	*Propagator+Battleship_BT (major), Evaluation, Report*

1. Project Motivation/Background

A Battleship Puzzle is a board of size *n* * *n*, with numbers along each row and column indicating the number of grids in this row or column that is occupied by a ship. A ship of size *l*, where *l* can take values from *1* to *n*, occupies *l* contiguous grids in a row or column without overlapping with other ships. Initially, the number of ships of each size is known. The goal of the puzzle is to **find a placement of the given battleships in the board that matches the number along the columns and rows**.

Inputs: three lists, *row_targets, col_targets, and ships*. Let *n* be board size. - ***row_targets***: a list storing the number of grids occupied by ship in each row - ***col_targets***: a list storing the number of grids occupied by ship in each column - ***ships***: a list storing the number of ships of size *1* to *n*, and *ships[i]* is the number of ships of size *i* (*ships[0]* is a dummy that is always *0*)	# sample problem **1 3 2 2** **2** ? ? ? ? **1** ? ? ? ? **2** ? ? ? ? **3** ? ? ? ? ships = [0,1,2,1]	# sample solution 0 0 2 2 0 2 0 0 0 2 0 1 3 3 3 0

2. Methods

The approach used is CSP together with custom backtracking search model 1. The initial thought is CSP since limits on rows and columns as well as the number of ships seem feasible to formalize as constraints. However, there were difficulty in building models (see section 4). As a result, the authors combined search, and introduced custom backtracking for model 1. Later on, the authors came up with model 2 and 3, which use traditional backtracking.

Description

1. Model 1 (*battleship_csp_model1*):
    a. **State variable:** each grid on the board is a variable
    b. **Variable domain:** the domain values are numbers in range *[0, n]*. *0* indicates no ship. Any non-zero value *l* indicates the grid is occupied by a ship of length *l*.
    c. **Constraints**:
        i. ***row_constraint***: the number of variables with value *>0* in each row *i* should match the number alongside row *i*, i.e. *row_targets[i]*. The scope of this constraint is each variable in the row.
        ii. ***col_constraint:*** symmetric to row_constraint.

    We will generalize the row and col constraints to be line constraints below.

    d. **Custom backtracking** (*battleship_BT.py*)

    Instead of assigning to 1 variable at a time in general CSP models, the custom BT search assigns multiple variables, i.e. assigning 1 ship at a time.

    In order to achieve this, the search starts from top-left corner of board, first assigning ships heading right, followed by assignments of ships heading down. This greatly **saves time assigning to neighbor variables values that cannot form a legal ship**, as in model 2. Moreover, it **avoids the need for constraints to check for overlapping ships on board**, since we keep track of currently assigned ships and do not allow assigning overlapping ships if grids are already occupied.

    However, this method makes **variable ordering heuristics useless**, since we have to choose variables in a top-left to bottom-right order.

e. **Space complexity**:

Let n be the board size, d be variable domain size ($d = O(n)$).

* *Variable space complexity* = *# of vars * var domain size*

$= O(n^2) * O(d) = O(n^2) * O(n) = O(n^3)$ *(polynomial)*

* *Constraint space complexity* = *# of line constraints * # of sat_tuple per constraint*[1]

$= O(n) * O(d^n) = O(n) * O(n^n) = O(n^{n+1})$ *(exponential)*

2. Model 2 (*battleship_csp_model2*):
   a. **State variable:** each grid on the board is a variable
   b. **Variable domain:** the domain values are tuples in the form *(ship_length, ship_id)*. *ship_length* is the ship length in the range *[0, n]*; *0* indicates the ocean, or no ship. Non-zero values indicate the length of the ship occupying that grid. *ship_id* is a unique number given to a ship in the range *[0, ships[ship_length]]*, in order to distinguish between ships of the same length for ease of ship overlap checks.
   c. **Constraints:**
      i. **row_constraint:** as in model 1, except value*>0* is now *ship_size >0*.
      ii. **col_constraint:** symmetric to row_constraint.
      iii. **ship_constraint:** the scope is all variables. Two checks included:
         1. Ship_num: total number of ships of length *l* matches the input *ships[l]*.
         2. Ship_intact: ships should occupy contiguous number of grids, i.e. the ship should not be broken[2].
   d. **Space complexity:**

Let n be the board size, d be variable domain size ($d = O(n^2)$).

Variable space complexity = *# of vars * var domain size*

$= O(n^2) * O(d) = O(n^2) * O(n^2) = O(n^4)$ *(polynomial)*

Constraint space complexity

*= # of line constraints * # of sat_tuple per constraint +*

*# of ship constraints * # of sat_tuple per constraint*

$= O(n) * O(d^n) + O(1) * O(d^{n^2}) = O(1) * O(d^{n^2}) = O(n^{2n^2})$*(exponential)*

3. Model 3 (*battleship_csp_model3*):
   a. **State variable:** for each grid at coordinate *(x,y)*, a variable is a ship starting from *(x,y)* of valid length, either heading right or down.
   b. **Variable domain:** the set of domain values for each variable are *{0, l}*, *l* being the ship length, *1 <= l <= n*. *0* indicates the ocean, or no ship. *l* indicates the ship is assigned.

[1] Here we see the number of satisfying tuples as the space bottleneck for each constraint and discard considerations for other space requirements for ease of analysis.
[2] By introducing the ids for ships of each size, ships not overlapping with each other could be ensured by ship_intact constraint.

c. **Constraints:**

i. **row_constraint:** as in model 1. The scope is every variable that lies on any grid in the row. If the variable represents a ship heading right, a value assignment of *l* occupies *l* grids in that row. If the variable represents a ship heading down, a value assignment of *l* occupies only 1 grid in that row.

ii. **col_constraint:** symmetric to row_constraint.

iii. **ship_constraint:** the scope is variables of the same ship length. Check if the total number of variables assigned non-zero values matches the input *ships[ship_length]*.

iv. **cell_overlap_cons:** at most 1 variable in the variables touching the same cell can be assigned non-zero value, or else the cell is occupied by multiple ships, which is illegal.

d. **Space complexity:**

Let n be the board size, d be variable domain size ($d = 2$).

Variable space complexity $= \sum_{l=1}^{max\_ship\_length}$ *(var of length l) * var domain size*

$= (O(n^2) * O(n)) * O(1) = O(n^3)$ *(polynomial)*

Constraint space complexity

*= # of line constraints * # of sat_tuples per constraint +*

*# of ship constraints * # of sat_tuples per constraint +*

*# of cell constraints * # of sat_tuples per constraint*

$=O(n) * O(d^{n^2}) + O(n) * O(d^n) + O(n^2) * O(d^{n^2}) = O(n^2 2^{n^2})$*(exponential)*

Comparison

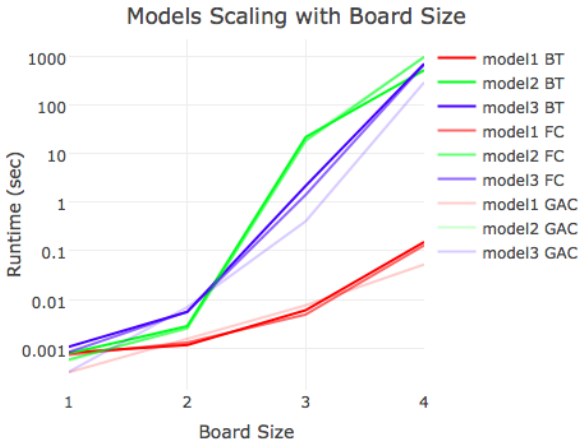
	Model 1	Model 2	Model 3
Type	CSP + Search	CSP	CSP
Runtime	Fast. Fast fast.	Slow. Stuck in BT search.	Slow. Stuck in model construction.
Space	Exponential.	Exponential. Also exponentially larger than model 1.	Exponential. Even larger than model 2.
Advantage	Efficiency.	Easy to implement.	Avoids useless assignments as in model 2 and improves efficiency.
Drawback	1. Customizing BT search is error-prone. 2. Variable ordering heuristics are useless.	1. Takes time finding satisfying tuples for each constraint due to large scope and domain. 2. Many useless assignments since neighbor variable assignments may not even form a legal ship.	Takes more time than model 2 forming satisfying tuples for each constraint due to big scope and many constraints.

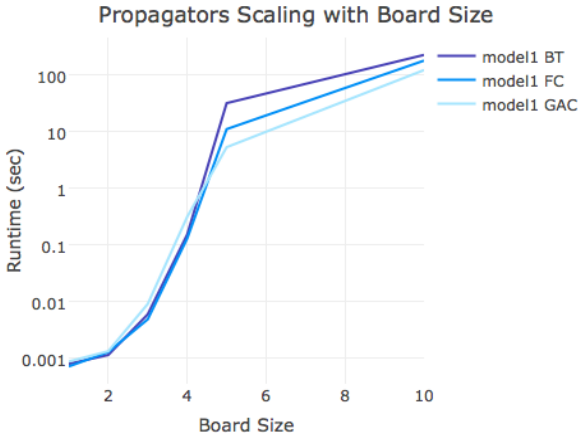
3. Evaluation and Results

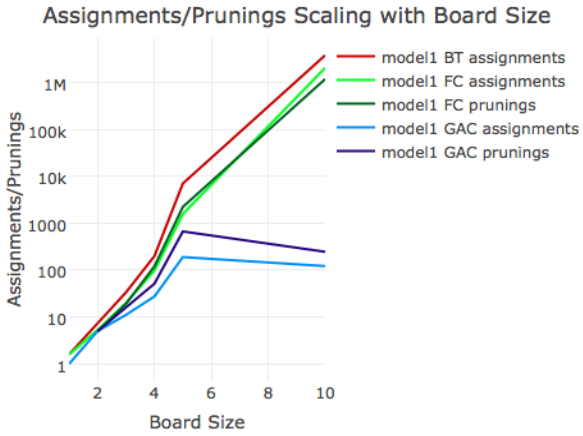
Since we have 3 CSP models, 3 propagation methods, 3 variable ordering heuristics and 5 value ordering heuristics, testing the performance for all possible combinations on different board sizes and target sizes was extremely resource intensive and difficult. In order to collect results on each configuration, we had 6 attempt runs on CDF servers, initially with loop iterations (simplest way to implement) and later to multithreading (which doesn't work so well as Python has a Global Interpreter Lock (GIL) that synchronizes the threads so that only one native thread can execute at a time) and finally to using multiprocessing, which worked really well, except when some of our test cases took more than 3 hours to solve, our test would be bottlenecked and could fail due to Python multiprocessing queue failing.

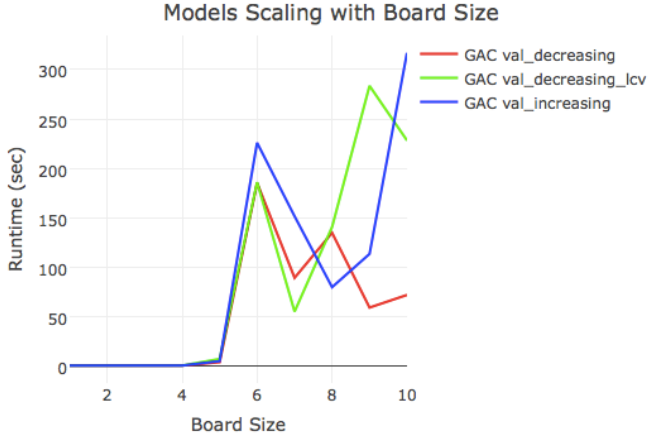
To further optimize our testing, we also tried to find out runtime patterns from our data and limit the test to only a specific sets of configurations so we could focus on a specific sets of configurations when we perform testing on larger board sizes (from 5x5 to 10x10 boards) instead of having to spend time testing inefficient models and ordering heuristics.

The results are as follow:









Graph Top-Left: This graph serves as an evidence that the performance of model 1 significantly outperforms the other two models. Model 2 is slow since it has a bottleneck in backtracking search due to the constraint formalization. Model 3 is slow due to the bottleneck in model construction. The performance difference becomes greater as the board size increases.

Graph Top-Right: For model 1, we see that in smaller board sizes, BT and FT had less overhead due to smaller number of checks required. It is until we move above 5x5 boards when GAC shows value in pruning domains and effectively shortening runtime.

Graph Bottom-Left: GAC has great effect on saving the variable assignments, and the number of assignments pruned is also reduced. The effect becomes more evident as board size grows.

Graph Bottom-Right: The trend is harder to tell since the testing server had different workloads from time to time and the runtime results are varying. However, assigning larger values first tends to reach the goal faster. This is somehow similar to greedily placing the ships on board, and by placing larger sized ships earlier, it is easier to exceed the limit of row/column targets and will avoid further advancements earlier.

Overall, pure CSP models perform not so well, while a little integration of search improves the performance greatly. Moreover, the advantage of GAC over the other two propagators becomes more evident as problem size grows. Finally, due to the nature of the game, we can see value in greedy ship placement.

4. Limitations/Obstacles

The major difficulty was model construction. Intuitively we encoded the variables in model 1 and 2 to be the grids on the board. In this case, **placements of the ship are not done in ship granularity** if we assign one variable at a time. As a result, the constraint to check ships overlapping and taking contiguous grids became difficult to formalize. Also, since the scope of this constraint was the set of all variables, the efficiency was undesirable.

Therefore, the authors introduced *battleship_BT* for model 1 to assign one ship (potentially more than one variable) at a time so that the constraint encoding is made simpler. In model 2, the numbering is introduced to make the overlap checking easier to implement. In model 3, we encoded ships as variables so that traditional backtracking (where one variable is assigned at a time) could be used.

5. Conclusions

The performance of pure CSP models are limited without intelligent model formalization. The bottleneck sometimes not only lies in the backtrack search but also in the model construction if naive satisfying tuple creation is used. Even if we identified the limitations in model 2 and did some improvements in model 3, the improvement is still limited.

Hence integrations of searching methods with CSP to overcome its limitations should be strongly advised. Evidence is shown in the performance of model 1. However, we did not have time to design more heuristics to further enhance the power of search. Some other methods can also be considered in the future, for example, after randomly placing the ships onto the board, move the ships around to approach the goal of reaching the requirements of row/column targets; many kinds of heuristics can be developed for this method.

In addition, tests can be written in advance to implementations. Our original design of the model was too naive and wasn't aware of special cases such as broken ship placements. With test cases generated first, we can save time heading in the wrong direction of implementation.

Finally, it would be better to analyze the time and space complexities of each method beforehand as well. We did not notice the bottleneck in model construction at first; it was until we finished and ran the program that we realized the excessive time spent in this step. Analysis of the complexities facilitates the comparison between different models, which can benefit in early identifications of a better method.