Foot Note 320 by Ian Gent
Limited Circulation[1] July 9, 1998

# Battling Battleships: A (failed ?) experiment in dual representation in Solver

This note documents a painful week I have spent battling the battleships problem in Solver. While some interesting issues have come up, my implementation has been full enough of bugs, some inexplicable to me, that I need to put it aside for now.

## 1 Battleships

The Battleships puzzle is occasionally published in *Games* Magazine (a US publication), e.g. in the August 1998 issue where it is credited to Mark Gottlieb. It is loosely based on the two person pencil and paper game.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | 0 |
| | | | | | | | | | | 2 |
| | | | | | | | | | | 3 |
| | | | | | | | | | | 1 |
| | | | | | | | | | | 2 |
| | | | | | | | | | | 4 |
| | | | | | | | | | | 2 |
| • | | | | | | | | | | 1 |
| | | | | | | | | | | 2 |
| | | | | | | | | | | 3 |
| 1 | 3 | 3 | 1 | 5 | 1 | 2 | 4 | 0 | 0 | |

I quote the puzzle description in the magazine:

> "This fleet consists of one battleship (four grid squares in length ••••), two cruisers (each grid squares long •••), three three destroyers (each two squares long ••) and four submarines (one square each •). The ships may be oriented horizontally or vertically, and no two ships will occupy adjacent grid squares, not even diagonally. The digits along the right side of and below the grid indicate the number of grid squares in the corresponding rows and columns that are occupied by vessels.
>
> In each of the puzzles, one or more "shots" have been taken to start you off. These may show water (indicated by wavy lines), a complete submarine (a circle), or the middle (a square), or the end (a rounded-off square) of a longer vessel. The puzzles get harder as you go."

The puzzle is naturally combinatorial in nature, and solving the above puzzle seemed to involve some search. That is, I could not see how to do any propagation beyond a certain point, and used some informed trial and error. So it was an obvious puzzle to try to represent in Solver. The above puzzle is number 2 of 6. The first puzzle did seem to be soluble without search.

## 2 Encoding using Dual Representation

When thinking about encoding a problem like this, there are typically two natural choices of variables and values. In this case the two obvious choices to me are:

- the squares are variables, the value being a 1 if occupied, 0 if not
- the ships are variables, with a twofold value being the starting square and orientation.

Actually these two formulations are almost duals of each other. If we change the representation slightly we can see this.

- the squares are variables, the value being the ship occupying the square (or some value indicating water)

[1] Should not be distributed outwith the APES group or people who should obviously be allowed to see it. Obviously!

- the ships are variables, with the value being the set of squares the ship occupies.

Indeed in general terms, we can usually form the dual of one formulation. We take the variables of the dual to be the values of the original, and the value to be the set of variables in the original taking that value. In this case, since only one ship can occupy a square, we do not need squares to have sets as values.

The Battleships problem struck me as one of many in which some constraints are most naturally expressed in one representation, while others more naturally in the dual. For example, the row and column sums seem naturally to be constraints on the squares, while the shape of the ships is naturally a constraint on the set of squares which are the value of a vessel.

Accordingly, I decided to try to implement the Battleships puzzle in Solver using both formulations simultaneously. At the cost of expressing constraints to link the dual formulations, one should be able to express all constraints in a natural way.

Unfortunately sets are not handled very nicely in Solver, so I decided instead to use the start square/orientation representation for ships, while indeed using the ships as values for the squares.

# 3 Object Disoriented Programming

Given that one set of variables are used as values for the other set, and vice versa, I felt that a nice way of encoding the problem would be in an object oriented style. I would like to have a set of objects for the ships, and a set of objects for the squares. While some features would be in ordinary attributes (e.g. square x-y position or ship type), others would be constrained variables. For example, a square would have a constrained variable taking as value the ship object which occupies the square (or some object representing water.)

Solver has a type of variable called IlcAnyVar, which can have values of type IlcAny, where IlcAny can be any type at all. This seemed ideal, and so I set about coding in an object oriented style.

An example constraint was that for any square in position `_x,_y`, unless the ship occupying that square is not water (i.e. really a ship at all), then its start x-y position, its orientation and length should all be consistent with the known x-y position of the square.

This constraint came out like this:

```
 ((Ship*)_ship)->isWater()
||
  ( (Ship*)_ship->isVertical()
    && ((Ship*)_ship->getStartX()) == _x
    && ((Ship*)_ship->getStartY()) <= _y
    && ((Ship*)_ship->getStartY())
        + ((Ship*)_ship->getLength()) > _y
  )
||
  ( ! (Ship*)_ship->isVertical()
    && (Ship*)_ship->getStartY() == _y
    && (Ship*)_ship->getStartX() <= _x
    && (Ship*)_ship->getStartX() + (Ship*)_ship->getLength > _x
  )
```

To explain it briefly to non C++'ers, `||` is *or*, and `!` is negation. The `(Ship*)` is a typecast because the value of `_ship` is really just a pointer so we need to tell the compiler what type the object really is. Having done that, `->isVertical()` calls the `isVertical()` method of whatever object the `_ship` variable's value points to.

Unfortunately, you can't get anywhere near writing constraints like this in Solver. Although IlcAnyVar's are supposed to support an object oriented style, the one example in the Manual (Stable Marriage Problem in Chapter 5) uses a very convoluted method posting demons. The problem is basically that while an IlcAnyVar's values are pointers to objects, you can't write constraints involving those objects methods. The only way round this is to wait until the object is bound, when its value can be accessed. So in the Stable Marriage example, constraints are only posted when values of variables are set.

Ironically, Solver does have some smart constraints which deal with pointers as values, but only in the context of arrays. For example, if `i` is an integer variable representing an index in an array, `spouse`, the constraint `spouse[spouse[i]]` succinctly and successfully states that the spouse of the spouse of `i` is `i` itself.

Of course in the example above `i` is not a person with a spouse, but just a number indexing an array. But since this kind of constraint is so useful, I find myself using what I call "Object-disoriented programming". Instead of having an array of objects, each with some features given by integer variables, I have a separate integer variable array for each feature, and think of the array index as being the object.

To illustrate this, the above constraint can be expressed as follows in object-disoriented style, assuming that for square `i` I have said that `ship = _squares[i]`:

```
  _isWater[ship]
||
  ( _ships._isVertical[ship]
    && (_ships._startx[ship] == x)
    && (_ships._starty[ship] <= y)
    && (_ships._starty[ship] + _ships._length[ship] > y)
  )
||
  ( ! _ships._isVertical[ship]
    && _ships._starty[ship] == y
    && _ships._startx[ship] <= x
    && _ships._startx[ship] + _ships._length[ship] > x
  )
```

Here, `_ships` is a single object containing a number of fields, each one being an array of integer/boolean variables. While `ship` is an integer variable representing the ship in my mind, or in reality the index in the array. In comparing the two pieces of code, you'll see `_ships` corresponds to the typecast `(Ship*)`. The rest of the object oriented call is reversed to give the disoriented version. So `ship` now comes last instead of first.

Of course writing in this style throws away most of the software engineering advantages of object oriented programming. Maybe that's why I have had so much trouble getting my code going, or going well.

# 4 Problems with Dual Formulation

Given this style, I was able to encode the battleships problem. I will put the code on the APES ftp site but won't discuss the detailed constraints here.

Unfortunately I haven't actually got it to solve the problem as given in Section 1. I did finally manage to get it solve it by filling in the positions of all the ships except the three destroyers. So the code seems to be correct, but not searching well. I have not investigated search heuristics seriously, but since I can solve the problems by hand (even without the water of life) I kind of have the feeling that we should not need smart heuristics to solve the problem.

One messy problem I had with the dual formulation was getting the symmetry constraints right. If we only have squares as variables, we only care about the bit indicating occupancy. But if squares have ships as values, we need to break the symmetry between equivalent cruisers, destroyers, and submarines. Dealing with this is not intellectually taxing, but did trip me up a lot in implementation.

More seriously though, I feel that the style of formulation I used can sometimes inhibit important constraint propagation. For example, suppose we know that a given square is occupied *either* by the battleship or by cruiser number 1. Of course this greatly limits the squares that the ship can start in. The constraint given earlier, in either form, cannot reduce the domain of either the battleship or the cruiser's start position, because whichever one is not occupying the square is unconstrained. However, we do have strong constraints on 'whatever ship occupies this square', because we know that it is either 3 or 4 squares long. This might tell us, for example, that whichever ship occupies the square must be horizontal. In turn we might deduce the actual start square. And finally we might be able to work out which of the two ships must really occupy the square. Unfortunately, because we have to shuttle back and forth between the two representations, we may have crippled the propagation enough to have prevented this deduction from happening.

In terms of my code, one can express fairly concretely what the problem is. Supposing that `i` is the square in question, and we have `ship = _squares[i]` In the situation described above, we might be able to constrain `_ships._startx[ship]` very tightly. But formally, `_ships._startx[ship]` is merely an IlcIntExp, i.e. an integer constrained *expression*. As such it does not have its own domain, and so while `_ships._startx[ship]` can only take a small number of values, a later reference to `_ships._startx[_squares[i]]` would have all the values still in the domain of either the battleship or the cruiser.

The solution in Solver terms is not hard. One can introduce a new integer variable array, perhaps writing `_squareStartX[i] = _ships._startx[squares[i]]` for each `i`. Then we rewrite our constraints using this even more disoriented style. Perhaps:

```
  _squareIsWater[i]
||
```

```
  ( _squareIsVertical[ship]
    && (_squareStartX[ship] == x)
    && (_squareStartY[ship] <= y)
    && (_squareStartY[ship] + _squareLength[ship] > y)
  )
||
  ( ! _squareIsVertical[ship]
    && _squareStarty[ship] == y
    && _squareStartx[ship] <= x
    && _squareStartx[ship] + _squareLength[ship] > x
  )
```

In this style, one is essentially constructing a candidate ship with properties which the actual ship occupying the square must conform to. One can take advantage of the dual representation to express certain constraints, either on the real dual formulation or the candidate objects. Indeed, it's quite likely that in this kind of formulation one would not need the actual dual representation at all, thus avoiding the need for some symmetry constraints.

I tried to implement this style in an attempt to reduce search, but was coming up with such inexplicable bugs that I gave up and wrote this note instead.

# 5 Conclusions

I don't have concrete proposals to make out of my experience. I still think it is interesting to consider trying to maintain both a formulation and its dual simultaneously if constraints can easily be expressed in one way or another. There are problems, and one possible solution to this might be to construct a kind of dummy dual representation. This allows the easy expression of constraints involving the dual representation, but perhaps avoids some of the problems which otherwise inhibit propagation.