Constraint Programming Models for Solitaire Battleships

Barbara M. Smith

Cork Constraint Computation Centre, University College Cork, Ireland
b.smith@4c.ucc.ie

November 9, 2006

Abstract. The Solitaire Battleships puzzle is described, and a range of constraint programming models are presented. The aim in building these models was to solve difficult instances of the puzzle from CSPLib, without resorting to searching for the solution. These are instances that existing rule-based software for Solitaire Battleships, *Fathom It!*, cannot solve. The puzzle is quite a challenge to model successfully using CP; a basic model is presented that requires some search even for easy instances. The search effort is much reduced by using regular constraints, at the expense of an increase in run-time. Finally, some of the CSPLib instances have been successfully solved without search, by using shaving. Since *Fathom It!* already does something very like shaving, this seems a promising route to solving a greater range of instances.

1 Introduction

Solitaire Battleships (prob014 in CSPLib) is a puzzle loosely based on the two-person game of Battleships. It is based on a squared grid, representing an area of ocean. Published puzzles typically use a 10×10 grid, although this can be varied. The patch of ocean contains a hidden fleet that the player must find. From the CSPLib problem specification: "*This fleet consists of one battleship (four grid squares in length), two cruisers (each three grid squares long), 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 player is also given the row and column tallies, i.e. the number of occupied squares in each row or column, and a number of hints. Each hint specifies the state of an individual square in the grid: *water* (the square is empty); *circle* (the square is occupied by a submarine); *middle* (this is a square in the middle of either a cruiser or a battleship); *top*, *bottom*, *left* or *right* (this square is the end of a ship occupying at least two squares).

In CSPLib, there are 303 difficult instances provided by Moshe Rubin, developer of the *Fathom It!* software (`www.mountainvistasoft.com`). *Fathom It!* provides a set of instances, of graded difficulty, each constructed to have exactly one solution. Figure 1 shows the starting state for an easy puzzle in *Fathom It!* The hints are *circle* in squares A10 and H9, *middle* in square D2 and *water* in square E9.

The documentation says that "The object in *Fathom It!* is to solve the board by locating the underlying fleet of ships using logical deduction only – no guess work!"

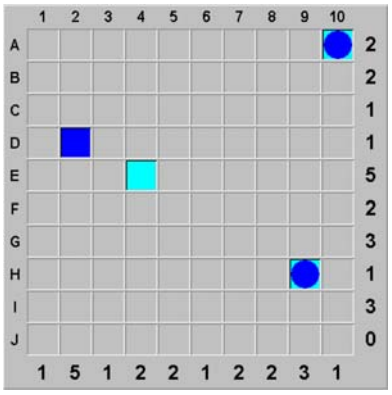

Fig. 1. Starting state for an easy Solitaire Battleships puzzle

The graphical interface allows the user to work towards solving the puzzle by making deductions (or guesses) and using the software to confirm their correctness. There is also an underlying rule-based system that will offer a deduction, given the current state of the board (including both the original hints and confirmed guesses), with an explanation of the reasoning leading to the deduction. By repeatedly asking for a suggestion and applying it, the player can use *Fathom It!* to solve the puzzle completely. The puzzles are graded as easy, intermediate, hard and expert levels, apparently based on the complexity of the rules that have to be used to solve them.

The 303 instances in CSPLib are puzzles that *Fathom It!* cannot solve; that is, at some point, it cannot offer the player any help in finding the solution. (It is important to note that the solution is known from the start; it is found in constructing the instance. Hence, the software can support the player by accepting or rejecting the player's guess. What it cannot do, for these instances, is explain how to find the solution by deduction rather than guessing.) Figure 2 shows the first instance in the set (instance 113), at the point where the software can make no further deductions; the original hints for this instance are that there is a submarine in G10 and that square A6 is water.

The Solitaire Battleships problem can be modelled as a constraint satisfaction problem, and the solution to a given instance found by search. Although this is an interesting exercise from the CP modelling point of view, it does not show how *Fathom It!* could be improved to solve the 303 CSPLib instances. However, if the solution to the CSP can be found purely by constraint propagation, without any search, for some of the CSPLib instances, that might suggest how *Fathom It!* could be improved. This paper describes the development of a CP model for Solitaire Battleships with this aim in mind.

2 A Basic Model

The first objective is to devise a simple CSP model that can represent the Solitaire Battleships problem correctly, even if requires more search than desirable. The row and column tallies suggest that the CSP should represent whether or not a square is occupied by a ship segment. The model also needs to be able to make deductions about the

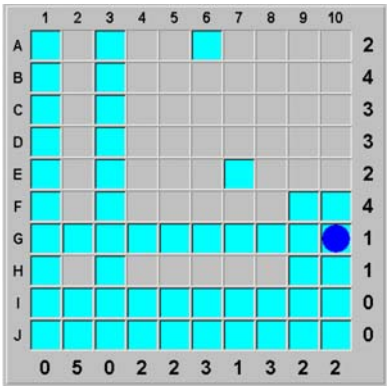

Fig. 2. A difficult Solitaire Battleships puzzle at the point where *Fathom It!* can make no further suggestions.

types of ships occupying various squares. It is perhaps tempting to explicitly represent the ends of ships, especially since some of the hints give this information, but in fact in solving the puzzles, the positions of the ends of ships tend to be decided as a consequence of other deductions. It is also tempting to specifically represent the positions of the battleships, cruisers, etc., but although this might ultimately be desirable, it is not obvious how best to do it, and how to deal with the symmetry between the two cruisers, for instance. For a simple model, it is sufficient to represent the type of an occupied square (battleship, cruiser, etc.); this allows all the necessary constraints to be expressed.

Let the size of the grid be $n \times n$, so that typically $n = 10$. For each square of the grid, and for a surrounding border one square wide, variables s_{ij} and t_{ij} are defined:

- s_{ij} = 1 if square (i, j) is occupied by a ship segment and 0 otherwise, $\forall i, j, 0 \leq i, j \leq n + 1$.
- $t_{ij} = 0$, if square (i, j) is unoccupied, or 1, 2, 3, 4 if the square is occupied by (part of) a submarine, destroyer, cruiser or battleship, respectively, $\forall i, j, 0 \leq i, j \leq n+1$. (Note that the type of a ship is the same as its length.)

The primary constraints on these variables are:

- $s_{0,j} = s_{n+1,j} = s_{i,0} = s_{i,n+1}$, $\forall i, j, 0 <= i, j \leq n + 1$.
- $\sum_j s_{i,j} = R_i$, where R_i is the tally (number of occupied squares) for row i; similarly, $\sum_i s_{ij} = C_j$, where C_j is the tally for column j.
- if $s_{ij} = 1$, then $s_{i-1,j-1} = s_{i-1,j+1} = s_{i+1,j+1} = s_{i+1,j-1} = 0$.
- channelling constraints: $s_{ij} = (t_{ij} > 0)$.
- a global cardinality constraint ensures that:

$$|\{t_{ij} | t_{ij} = k, 1 \leq i \leq n, 1 \leq j \leq n\}| = l$$

where l = 4, 6, 6, 4 when k = 1, 2, 3, 4 respectively, i.e. the number of squares occupied by submarines is 4, the number of squares occupied by destroyers is 6, and so on.

The model is not yet complete, since there is nothing to ensure that $t_{ij} = p, p > 0$, iff the square (i, j) occurs in a run of exactly p occupied squares. Auxiliary *ladder* variables [3] can be introduced to represent how far the stretch of occupied squares adjacent to an occupied square runs in each direction. The Boolean variables $r_{ijk}, 1 \leq k \leq 4$ indicate whether there is a run of occupied squares from (i, j) to $(i, j + k)$. If $r_{ijk} = 1$, the square (i, j) is occupied by a ship that also occupies the square $(i, j + k)$, and of course all intervening squares, if any.

The constraints on the ladder variables are:

- $r_{ij1} = 1$ iff $s_{ij} = 1$ and $s_{i,j+1} = 1$.
- $r_{ij,k+1} = 1$ iff $r_{ijk} = 1$ and $s_{i,j+k+1} = 1$ for $1 \leq k \leq 3$ and $j + k \leq n$.

There are similar sets of ladder variables $l_{ijk}, u_{ijk}, d_{ijk}, 1 \leq k \leq 4$, for the squares to the right, above and below the square (i, j), and the constraint ensuring the correct value of t_{ij} is then:

$$t_{ij} = \max(\sum_k r_{ijk} + \sum_k l_{ijk}, \sum_k u_{ijk} + \sum_k d_{ijk})$$

An initial hint relating to a square (i, j) is represented as constraints as follows:

- *water*: $s_{ij} = 0$.
- *circle*: $s_{i-1,j} = s_{i+1,j} = s_{i,j-1} = s_{i,j+1} = 0$.
- *left*; $s_{i-1,j} = s_{i+1,j} = s_{i,j-1} = 0, s_{i,j+1} = 1$ (and similarly for *right*, *top* and *bottom* hints).
- *middle*: $t_{ij} \geq 3$ and either $s_{i-1,j} = s_{i+1,j} = 0$ and $s_{i,j-1} = s_{i,j+1} = 1$ or $s_{i-1,j} = s_{i+1,j} = 0$ and $s_{i,j-1} = s_{i,j+1} = 1$.

3 Search Heuristics

Ideally, the constraints expressing the problem should be sufficient to allow the unique solution to the puzzle to be found without search, simply by propagation. However, with the constraints described so far, constraint propagation is not sufficient, even for easy instances, and so the solution has to be found by search.

The search uses the default backtracking search algorithm in ILOG Solver 6.0. The search variables could be either the *ship segment* variables, s_{ij}, $1 \leq i, j \leq n$, or the *type* variables, t_{ij}, $1 \leq i, j \leq n$. Clearly, any solution to an instance can be expressed as a complete assignment to either set of variables. (Conceivably, both sets of variables could be used as search variables at the same time, but this has not been explored.)

It is not obvious which set of search variables is the better choice, so both have been tried, with several different search strategies. For the *ship segment* variables, lexicographic variable ordering was used, considering squares by row, from top to bottom, and within each row, from left to right. Since these are Boolean variables, smallest-domain variable ordering would reduce to lexicographic ordering. For the *type* variables, smallest-domain variable ordering is an obvious choice; lexicographic ordering was also used. An alternative strategy, based on the intuition that it may often be a good idea to decide where to put the battleship first, then the cruisers and so on, is to choose

next the variable with the largest value in its domain, using smallest-domain as a tie-breaker. Following the same intuition, having chosen a variable with the largest value in its domain, the value assigned to it should be that value. Hence, two value-ordering strategies have been tried: choosing the smallest value in the domain, or the largest. These have been combined with all the previous choices, giving a total of eight ways of solving each instance.

4 Experimental Results: Basic Model

The CP model, with the eight possible search strategies, has been applied to three sets of instances of Solitaire Battleships. The first set consists of the ten easy instances in the evaluation version of *Fathom It!*; in the second are the ten intermediate instances from the same source. The third is the set of 303 instances from CSPLib. The model is implemented in ILOG Solver 6.0, running under Windows 2000, on a 1.7GHz Pentium M PC. For each search strategy and each set, the total number of backtracks to solve all the problems in each set was recorded, with a cut-off of 1 million backtracks, since the basic model has to do a lot of search to solve all the instances in the third set. The results for the basic model are shown in Table 1. With this model, the simplest search strategy beats the special-purpose variable and value ordering heuristics, and for the CSPLib instances, this is the only strategy that can solve them within the cut-off. The run-time is not given, since the principal aim is to be able to solve each instance with as little search as possible.

Table 1. Total number of backtracks to solve instances of Solitaire Battleships, using the basic CP model.

Search Variables	*ship segment*		*type*					
Variable ordering	lexicographic		lexicographic		min. domain		max. value + min. domain	
Value Ordering	min	max	min	max	min	max	min	max
Easy instances	112	75	232	687	322	1,061	217	1,709
Intermediate instances	983	1,294	4,718	4,150	5,729	5,680	7,470	7,407
CSPLib instances	348,812	409,392	$>$ 1m.	$>$ 1m.	$>$ 1m.	$>$ 1m.	$>$ 1m.	$>$ 1m.

5 Improving the model: Implied Constraints

Although the CP model already described is correct, and the solution to each instance can be found, in most cases quite quickly, it requires some search to find it, even for some of the easy instances.

By examining the domains of the type variables, t_{ij}, at points where the search is forced to make a choice, situations can be found where the domains could be pruned but the existing constraints do not allow it. The model can then be modified to allow the correct deductions to be made.

– In the situation shown below, the value 2 has been assigned to the centre square, while the two squares to either side of it have domains $\{0, 1, 2, 3\}$.

0..3	2	0..3

It is clearly impossible to have a square of type 1 or 3 next to a square of type 2, so that the domains could be pruned to $\{0, 2\}$. This can be done by adding the constraints:

if $s_{ij} = 1$ then $t_{i-1,j} = t_{ij}$ or $t_{i-1,j} = 0 \quad \forall i, j$ with $1 \leq i, j \leq n$

and similarly for $t_{i+1,j}$, $t_{i,j-1}$, $t_{i,j+1}$.

– Search states can occur where the value 4, say, occurs in the domain of a type variable, but clearly a battleship cannot be placed in that row or column because the relevant tally does not allow it. A constraint to forbid such a situation is:

$$t_{ij} \leq \max(R_i, C_j) \;\; \forall i, j \text{ with } 1 \leq i, j \leq n$$

– In the following run of squares of type 4, only one more square can be added to the battleship (and indeed must be): the next square must then be empty.

0	4	4	4	0..4	0..4	0..4

A simple set of constraints to eliminate this situation are: $l_{ij4} = r_{ij4} = u_{ij4} = d_{ij4} = 0, \forall i, j$ with $1 \leq i, j \leq n$.

Table 2 shows the effect on total search effort of adding these implied constraints to the basic CP model. The search effort is reduced considerably, especially for the harder problems, and the special-purpose search strategy is now more successful.

Table 2. Total number of backtracks to solve instances of Solitaire Battleships, using the basic CP model + implied constraints.

Search Variables	*ship segment*		*type*					
Variable ordering	lexicographic		lexicographic		min. domain		max. value + min. domain	
Value Ordering	min	max	min	max	min	max	min	max
Easy instances	33	27	41	42	27	134	40	8
Intermediate instances	738	728	918	834	1,549	1,355	362	189
CSPLib instances	278,392	301,561	418,974	619,762	345,304	$>$ 1m.	212,395	341,610

5.1 Regular Constraints

Although the implied constraints reduce the number of backtracks required to solve the instances, Table 2 shows that even the easy instances cannot yet be solved without search.

So far, little use is made of the row and column tallies, apart from insisting that the type of a square is no greater than the maximum of its row and column tallies.

The configuration below, showing the domains of the type variables in a row of the table, is perfectly acceptable in itself. However, if the row tally is 3, there is no room for a cruiser (type 3) on this row as well as the part of a destroyer (type 2) that is already assigned to it. Consequently, the value 3 can be pruned from the domains.

0..1	0	0..3	0..3	0..3	0	2	0	0..2	0..2

In the example, there is a destroyer occupying part of the row, but because it is oriented column-wise, there is room for another destroyer oriented row-wise in this row. More variables have been added to the model, to represent the *orientation* of the ship in an occupied square: $o_{i,j}$ = 1 if there is a ship of type ≥ 2 oriented column-wise in square (i, j), 0 otherwise.

The configurations of ship segments that are possible in a row or column of the board, with their ship types and orientations, can be defined using a *regular* constraint [4]. The regular constraint is a global constraint on a sequence of finite-domain variables that ensures that the sequence of values taken by the variables forms a string accepted by a given deterministic finite automaton (DFA). Quimper and Walsh [5] point out that a regular constraint can be encoded by a sequence of ternary constraints and that generalized arc consistency (GAC) on the ternary constraints achieves GAC on the regular constraint. The regular constraint on row i will be described in detail; there are similar constraints on the columns.

To state the allowed sequences for a given row i, we introduce variables $X_{i0}, X_{i1}, ..., X_{in}, X_{i,n+1}$, where X_{ij} represents the tuple $\langle t_{ij}, o_{ij}\rangle$. The encoding requires a second sequence of variables $S_{i0}, S_{i1}, ..., S_{i,n+1}$ representing the corresponding state of the DFA.

To state the constraint requires specification of a set of states and a transition function. In this case, the possible values of X_{ij}, considered as the next item in the sequence $X_{i0}, X_{i1}, ..., X_{i,j-1}$, depend on several aspects of the sequence of values so far. The variable S_{ij} is a 3-tuple composed of:

- X_{ij};
- p_{ij}, where p_{ij} is the number of occupied squares in the sequence $(i, 0), (i, 1), ..., (i, j)$;
- q_{ij}, where $q_{ij} = 0$ if square (i, j) is unoccupied and otherwise is the length of the run of occupied squares in this row, up to and including square (i, j).

The initial state is $\langle 0, 0, 0\rangle$ and the set of accepting states for the DFA consists of the single state $\langle 0, R_i, 0\rangle$, where R_i is the row tally.

The regular constraint can then be encoded by ternary transition constraints on the variables $X_{i,j+1}, S_{i,j}, S_{i,j+1}$ for $0 \leq i \leq n+1$ which hold iff $S_{i,j+1} = T(X_{i,j+1}, S_{ij})$

where T is the transition function of the DFA, and the unary constraints $S_{i0} = \langle 0, 0, 0 \rangle$ and $S_{i,n+1} = \langle 0, R_i, 0 \rangle$.

The ternary constraints are represented in ILOG Solver by a table constraint defined by a predicate returning *true* for any combination of values for $S_{i,j+1}, X_{i,j+1}, S_{ij}$ which is allowed and *false* otherwise. This allows the predicate to deal separately with the components of the tuples. First, we can state acceptable combinations of values for the variables $p_{ij}, p_{i,j+1}, q_{ij}, q_{i,j+1}$, for $0 \leq j \leq n$, depending on the value of $t_{i,j+1}$:

- if $t_{i,j+1} = 0$ (i.e. square $(i, j+1)$ is empty), we must have $p_{i,j+1} = p_{i,j}$ and $q_{i,j+1} = 0$ (i.e. the number of occupied squares in the sequence so far is unchanged and the number of occupied squares in the current run is 0);
- if $t_{i,j+1} > 0$ (i.e. the square is occupied by a ship segment), we must have $p_{i,j+1} = p_{i,j} + 1$ and $q_{i,j+1} = q_{i,j} + 1$.

It remains to specify the possible combinations of values for $X_{i,j+1}$ and $X_{i,j}$, for $0 \leq j \leq n$. The simplest case is when $t_{i,j} > 0$, i.e. square (i, j) is occupied. Then square $(i, j+1)$ must be occupied by a ship of the same type, if it is a ship placed row-wise and is not yet complete, and otherwise must be empty:

- if $o_{ij} = 1$, i.e. there is a ship placed column-wise in square (i, j), then $t_{i,j+1} = o_{i,j+1} = 0$;
- if $o_{ij} = 0$ and $t_{ij} = q_{ij} > 0$, i.e. there is a ship placed row-wise in square (i, j) but it is now complete, then $t_{i,j+1} = 0$;
- if $o_{ij} = 0$ and $t_{ij} < q_{ij}$, then $t_{i,j+1} = t_{ij}$ and $o_{i,j+1} = o_{ij}$.

If square (i, j) is empty, the predicate has to recognise when square $(i, j+1)$ cannot be occupied at all, because every ship segment in this row has already been placed ($p_{i,j} = R_i$), or cannot be occupied by a ship of length l placed row-wise ($p_{i,j} + l > R_i$). It should also recognise when square $(i, j+1)$ *must* be occupied. This depends on the minimum number of squares, s_{min}, needed to accommodate the remaining ship segments in this row, $R_i - p_{ij}$. If the remaining ship segments could be a single ship ($0 < R_i - p_{ij} \leq 4$), then $s_{min} = R_i - p_{ij}$; if at least two ships are needed($4 < R_i - p_{ij} \leq 7$), then $s_{min} = R_i - p_{ij} + 1$, to allow for an intervening square between the two ships; if more than two ships are needed ($R_i - p_{ij} > 7$), then $s_{min} = R_i - p_{ij} + 2$. For boards of size 10×10, this is sufficient, because the maximum row tally is 8. Square $(i, j+1)$ must be occupied if the number of squares remaining in the row, $n - j$, is equal to s_{min}.

Enforcing GAC on the ternary constraints expressed by the predicate just described ensures that within each row, values in the domains of the variables that cannot be extended to a valid sequence are pruned. Although the constraint is complicated to describe, the domain deletions that it leads to are in practice easy to understand. For a human player, it is difficult to keep track of the possible types of ship that could occupy each square, and so the propagation of the regular constraints (and of course the type variables of the CP model) do not match how a human player solves the puzzle. However, propagation of the regular constraint appears to lead to the same deductions about squares that must or must not be occupied that a competent human player could make.

The regular constraints supersede some parts of the earlier CP models; the ladder variables are no longer needed, nor are the implied constraints described in section 5.

Table 3 shows the experimental results from the CP model with regular constraints. The easy instances can now be solved without search; propagating the constraints is sufficient to identify the solution. However, at the next level of difficulty, nearly all the intermediate instances still require search. For the CSPLib instances, the total search effort, as measured by number of backtracks, decreases by more than 85%, compared to the basic model with implied constraints, with max value/min domain variable ordering, choosing the largest value. However, propagating the regular constraints is time-consuming, since the state variables have large domains, and so the total run-time increases from 125 to 165 seconds.

Table 3. Total number of backtracks to solve instances of Solitaire Battleships, using regular constraints on the rows and columns.

Search Variables	*ship segment*		*type*					
Variable ordering	lexicographic		lexicographic		min. domain		max. value + min. domain	
Value Ordering	min	max	min	max	min	max	min	max
Easy instances	0	0	0	0	0	0	0	0
Intermediate instances	226	296	264	255	180	313	41	7
CSPLib instances	124,913	131,399	131,207	138,901	142,733	237,260	44,424	41,601

6 Shaving

Fathom It! can make suggestions that are reminiscent of singleton arc consistency or shaving. For example, one of suggestions given by *Fathom It!*, with the accompanying explanation, is:

> Square E7 must be water.
> Reasoning: if it were a ship segment, we would have the following contradiction:
> In row F you must place 4 more ship segments, but have only 3 free squares.

In terms of the CP model, assigning the value 1 to variable $s_{5,7}$ results in a contradiction, so that value must be removed from the domain of the variable, and hence it must take the value 0. Implementing shaving could therefore improve the performance of the CP model and bring it closer to that of *Fathom It!*

Applying shaving to the Boolean variables s_{ij}, that indicate whether or not a square is occupied by a ship segment, is most similar to the way that *Fathom It!* operates. Shaving is done as a preliminary step, before starting search (if that is still necessary). For each square (i, j), if either forcing this square to be a ship segment or forcing it to be empty results in a contradiction, the corresponding variable s_{ij} is set to 0 or 1 respectively.

With this form of shaving, the 10 intermediate instances can all be solved without search. More significantly, 15 of the 303 CSPLib instances can also be solved without search. Of these, one instance could not be solved by *Fathom It!* because of a bug that had not previously been recognised; another two can be solved by the current version of the software. The remaining 12 instances remain unsolvable for *Fathom It!*: it reaches a point where it cannot offer any further suggestions. The CP solver does not offer any explanation of its actions, apart from listing the squares that it finds must be occupied or unoccupied, but it is not too difficult to see the contradiction by making the opposite decision and following the consequences; hence, it seems that it should be possible to offer these as suggestions to the player and provide an explanation. While solving the 303 CSPLib instances, shaving reduces the total number of backtracks from 41,601 to 38,715, using max value/min domain variable ordering and choosing the largest value. Shaving is time-consuming, and the total run-time increases again from 165 to 195 seconds.

A second form of shaving has also been implemented; in this case, if forcing square (i, j) to be a ship segment does not result in a contradiction, shaving is also done on the type variable t_{ij}, to see whether any deduction can be made about the possible ship types that can occupy the square. With this form of shaving, another 54 of the 303 CSPLib instances can be solved without search. However, it is harder to reconstruct the reasoning by hand; in many cases, both possible orientations for a ship of that type have to be shown to result in a contradiction. Moreover, it is harder to record and keep track of the possible ship types that can occupy a square than simply whether or not the square is occupied. Overall, the total number of backtracks to solve the CSPLib instances is reduced to 26,640, but the run-time increases to 350 seconds.

7 Further Improvements

7.1 Search Strategy

Although the ultimate aim in modelling Solitaire Battleships is to find the solution by constraint propagation rather than search, until that can be achieved, a subsidiary aim should be to minimise the search. Table 3 shows that the best search strategy depends on the model, providing further confirmation of an observation made by Beacham *et al.* [1]. With the basic model, searching on the ship segment variables, s_{ij}, is the better choice, whereas with the final model it is better to use the type variables, t_{ij}, as the search variables, and to use the search strategy which was devised to place the ships in descending order of size.

The ship segment variables might be thought to be a better choice *a priori*, because they entail less commitment than the type variables. On the other hand, lexicographic ordering cannot take into account the current state of the search. With the type variables, it is clearly the dynamic variable ordering that gives the advantage; lexicographic ordering on these variables is worse than lexicographic ordering on the ship segment variables.

It should be possible to devise a dynamic variable-ordering heuristic for the ship segment variables, based on the way in which *Fathom It!* applies its rules; for instance,

if there is a row with tally k and there are $k + 1$ undecided squares, and no squares yet occupied, a good plan is to choose to make one of the undecided squares unoccupied, hence making all remaining squares in the row occupied.

One possibility would be to use a look-ahead heuristic, like the *promise* value ordering heuristic [2] which assigns the value for the chosen variable that will leave the largest number of values for the unassigned variables. In Solitaire Battleships, a variable ordering heuristic for the ship segment variables might choose the variable-value combination that would result in setting the largest number of other ship segment variables to the value 1 (possibly using the largest number of ship segment variables set to 0 as a tiebreaker).

The suggested deductions given by *Fathom It!* are always whether or not a square is a ship segment; although the deduction is sometimes based on considerations of the location of different types of ship, it is often based purely on row and column tallies, combined with the number of occupied or unoccupied squares already decided. This supports the idea that a search strategy based on the Boolean ship segment variables could be successful.

7.2 Constraint Propagation

There is some way to go before the constraint programming models presented here can equal the performance of *Fathom It!* An obvious shortcoming is that there is very little reasoning about where the ships of various types can be; this is demonstrated by considering one of the instances that *Fathom It!* can solve and the CP model cannot. With shaving on the s_{ij} variables, the CP model can solve 5 the first 10 hard instances in the *Fathom It!* collection. Figure 3 shows the board state with the deductions due to shaving, together with their immediate consequences.

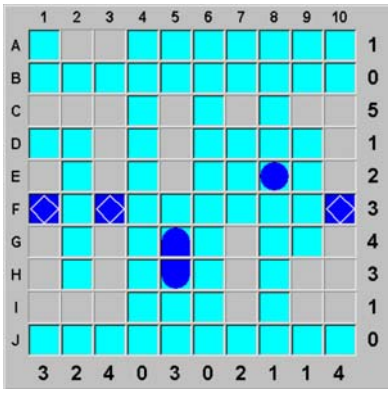

Fig. 3. Board state after shaving for a hard Solitaire Battleships puzzle

Given this state, *Fathom It!* offers the following suggestion (slightly paraphrased) as the first step towards completing the puzzle:

Square A2 must be a ship segment.
Reasoning: If it were water, we would have the following contradiction:
All squares remaining in column 2 are ship segments.
All squares remaining in row A are ship segments.
Since all 4 submarines are accounted for, the following empty squares can be filled with water: C7. Reasoning: There are 3 finalized submarines on the board. The remaining submarines are located as follows: 1 submarine along column 5.
All squares remaining in column 7 are ship segments.
Placing the battleship results in the following overlapped squares: E10, F10 (already finalized). Reasoning: There are 2 valid battleship positions on the board: C10 - F10, E10 - H10. There is 1 non-valid battleship position on the board: D10 - G10 (Closes off row C). The squares (E,10) and (F,10) overlap and are ship segments.
All squares remaining in column 1 are ship segments.
All squares remaining in column 3 are ship segments.
More than 3 destroyers are forced.

The chain of reasoning is easier to follow in *Fathom It!* because it is displayed on a copy of the board. This makes it easy to see, for instance, that the battleship cannot be in column 3, because the squares D3 and H3 must be unoccupied.

During the shaving step, the CP model tries forcing square A2 to be unoccupied, but is unable to derive a contradiction, because it cannot reason about the positions of the submarines and other types of ship as *Fathom It!* can. It is not clear what would be the best way to extend the model to allow this kind of reasoning. One possibility, for the battleship, is to introduce a set variable whose value is the set of squares occupied by the battleship. There are 140 possible positions for the battleship on a 10×10 board, so that would be feasible to keep track of all remaining possibilities as the board is completed. This would allow deductions of the kind shown above, where the remaining possibilities overlap. However, it would not allow the deduction that a given square must be water because it cannot be a ship segment for any possible placement of the battleship; *Fathom It!* can and does make deductions of this kind. Furthermore, using a single set variable to represent the squares occupied by a given type of ship would not be feasible for the other ship types. It seems that it would be necessary to represent, say, the two cruisers by different variables, although this would introduce symmetry into the model and so would also need some form of symmetry breaking.

8 Conclusions

Solitaire Battleships is a challenging problem for human players. It is also challenging for constraint programming, if the aim is not simply to find the solution but to find it with little or no search, and rely on constraint propagation. This would certainly be a necessary first step if we wanted to emulate *Fathom It!* and support the player by offering deductions that can be easily understood, with an explanation. The aim in developing successive CP models has therefore been to reduce the number of backtracks, even

at the expense of increased run-time. Although the aim is to do no search, some very limited search may be acceptable if necessary: occasionally, the explanations given by *Fathom It!* for the hardest instances that it can solve have other deductions, and explanations of those deductions, nested inside them, and so require something like search.

We can contrast Solitaire Battleships with Sudoku, for instance, where published puzzles similarly have exactly one solution. In that case, the problem is easily modelled by allDifferent constraints and very few instances require any search once the constraints have been propagated; Simonis [6] added shaving and could find no published instances which then required any search. The very different experience in Solitaire Battleships is not because constraint programming is an inappropriate technique, but because the problem itself is more complex.

A basic CP model which represents the problem accurately has been described; with this model, even easy instances could not be solved without search. A more complex model which incorporates a regular constraint on each row and on each column can solve these instances without search; with the addition of a shaving step applied to the variables representing whether or not a square is occupied, even some of the difficult instances in CSPLib that the *Fathom It!* software cannot solve are solved without search.

By shaving on the variables representing the type of ship occupying a square, many more of the CSPLIb instances can be solved. However, it is much more difficult to understand the domain deletions in this case, largely because there are more possibilities to keep track of. It would be very difficult to incorporate anything like this into *Fathom It!*; at present it has no way to record the possible types of ship in each square in its interface, and it is doubtful that this would be desirable.

Even though the CP model is not overall able to compete with *Fathom It!*, it has solved some of the CSPLib instances using shaving, in a way that is relatively easy for a human player to understand. Since *Fathom It!* already does something very like shaving on the ship segment variables, the experience with the CP model suggests that applying the relevant rules more widely would allow it to solve at least some of the CSPLib problems that it currently cannot solve.

Acknowledgments

I am grateful to Moshe Rubin for permission to use *Fathom It!* screen shots, and for a beta version of the software to help in investigating the problem. I should also like to thank Olivier Lhomme, Jean-François Puget and Paul Shaw for showing me how to implement shaving in ILOG Solver. This material is based in part on works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075.

References

1. A. Beacham, X. Chen, J. Sillito, and P. van Beek. Constraint programming lessons learned from crossword puzzles. In *Proceedings of the 14th Canadian Conference on Artificial Intelligence*, pages 78–87, 2001.
2. P. A. Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In B. Neumann, editor, *Proceedings ECAI'92*, pages 31–35, 1992.

3. I. P. Gent, P. Prosser, and B. M. Smith. A 0/1 encoding of the GACLex constraint for pairs of vectors. 2002. Presented at the ECAI'02 Workshop on Modelling and Solving Problems with Constraints. Available from http://www.dcs.st-and.ac.uk/ cppod/publications/gpsW9.pdf.
4. G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In M. Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004*, volume LNCS 3258, pages 482–495. Springer, 2004.
5. C.-G. Quimper and T. Walsh. Global Grammar Constraints. In F. Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume LNCS 4204, pages 751–755. Springer, 2006.
6. H. Simonis. Sudoku as a Constraint Problem. In *Proceedings of the CP05 Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2005.