Crossword Grid Composition with a Hierarchical CSP Encoding

Adi Botea

NICTA and
Australian National University
Canberra, ACT

Abstract. Automatic composition of crossword grids is challenging for many interesting puzzles. This paper introduces a solving approach that uses a hierarchical CSP encoding. At the high level, word slots are variables and all dictionary words that fit into that slot are possible values. The low level uses each grid cell as a variable that has all alphabet characters as possible values. Searching for a solution explores the high-level space, instantiating entire words rather than individual cells. Channelling constraints between the two levels reduce the high-level search space. The benefits of the new model are demonstrated with experiments on large puzzles.

1 Introduction

Crossword grid composition is both a problem that illustrates human intelligence and a textbook application of AI constraint programming. Despite its beauty and relevance to AI research, it has received limited attention from academic researchers, especially when compared to other puzzles and games. Automatic composition of crossword grids is challenging for a large class of interesting puzzles. Features that impact the complexity of a puzzle include the size of the grid, the size of the dictionary, the pattern of blocked cells and the percentage of blocked cells.

Consider, for example, varying the number of blocked cells. Problems with relatively many blocked cells (e.g., more than 15% on a 15x15 grid) have very many solutions and reaching a goal state is quite easy. At the other extreme, problems with no blocked cells have very few or no solutions. Complete exploration of the problem space of such tightly constrained puzzles is possible even with simple pruning rules based on deadlock detection.[1] Puzzles become much harder when the percentage of blocked cells shifts from these extremes towards the middle of the range.

A major performance bottleneck can be caused by the presence of deadlocks. Experiments on hard puzzles have shown that the same deadlock can occur over and over again, when changes in one corner do not affect the deadlock area. Being able to detect deadlocks early is crucial for the performance of a solver.

This paper presents a solving model based on a hierarchical CSP encoding of the problem. The higher hierarchical level defines one variable for each word slot. A low-level variable is created for each non blocked cell on the grid. Searching for a solution is

[1] A deadlock state is a partially filled grid for which no correct completion exists.

performed at the high level, instantiating entire word slots rather than individual cells. The low level is used to reduce the high-level search space via channelling constraints.

In this paper, a puzzle consists of a dictionary, a grid size, and a pattern of blocked cells. The task is to fill all word slots with valid dictionary words. No blocked cells can be added or removed during the solving process. The related topic of automatically solving puzzles based on their clue lists [2, 5] is beyond the focus of this paper.

The rest of the paper is structured as follows: The next section reviews related work. Then we present the hierarchical model in detail and discuss how it can be extended to handle any CSP problem, not only crossword composition. A brief section describes the strategy employed when searching for a solution. The empirical evaluation comes next, followed by conclusions and future work ideas.

2 Related Work

This section starts with a review of work on crossword grid composition. The last part focuses on related work in constraint programming.

While several commercial products are available, the AI literature provides few contributions to the topic of automatic composition of crosswords. In the early work of Mazlack [7] a grid is filled with a letter-by-letter approach. More recently, Ginsberg *et al.* [3] focus on an approach that adds an entire word at a time. The list of matching words for each slot is updated dynamically based on the slot positions already filled with letters. We advance this by computing more accurate word lists with an additional level of variables. Meehan and Gray [8] compare a letter-by-letter approach against a word-by-word encoding and conclude that the latter is able to scale up to harder puzzles.

Beacham *et al.* [1] use the crossword application as a testbed to study how choosing a combination of a problem encoding (which can be either CSP or SAT-based), a search strategy and a heuristic impact the performance of a solver. The CSP models include pure encodings where only word slots or only cells generate CSP variables, and a hybrid model where both slots and cells are variables. In the hybrid model, no distinction is made between the two types of variables. In contrast, our architecture is a combination of two *viewpoints* (i.e., mutually redundant encodings of a problem), each corresponding to one variable type. The connection between two viewpoints is achieved with a set of *channelling constraints*.

In constraint programming, combining two viewpoints into one model is by no means a new idea. See Smith's survey [9] for work in this area. Hnich *et al.* [4] combine several variables into a so called *compound variable*. The authors apply this idea to model the covering test problem. Structuring a CSP problem hierarchically has been used by Mackworth *et al.* [6]. In that research, the domain of a variable is represented as a hierarchy. In our work, the set of variables is partitioned into a high level and a low level.

3 Hierarchical CSP Encoding

Our model uses a dual CSP representation of a problem, at two different granularity levels. In the *high-level* representation each word slot is a variable whose possible values are all words that fit into that slot. As a *low-level* encoding, each non blocked cell introduces a variable that has all alphabet characters as possible values. As shown before, neither encoding is new. Our contribution is to combine them into a hierarchical model and exploit the strengths of each encoding. The main strength of the high-level encoding that we exploit in this work is a much smaller search space than the low-level encoding. The low-level encoding allows to introduce a set of channelling constraints that directly impact the branching factor of a node, the deadlock detection mechanism, and the decision of what grid area to explore next. The difference in search space size between the high-level and the low-level encodings is illustrated next. The impact of the low-level encoding on search is more elaborated and it is presented in the following sections.

An upper bound for the size of the low-level space is $N_c^{|\mathcal{A}|}$, where N_c is the total number of empty cells and $|\mathcal{A}|$ is the alphabet size. The high-level search space would approach this limit only if *all* letter sequences were valid English words. In practice, only a tiny subset of these sequences are real words. The number of real words of a given length is in the order of thousands or tens of thousands. In contrast, the total number of, say, 10-letter sequences is 10^{26}.

Before describing the hierarchical architecture in detail, we introduce some definitions and notations that will be used in the rest of the article. Slots (i.e., high-level variables) will be denoted with s. The length of a slot is $l(s)$, and its i-th cell is $s[i]$, $1 \leq i \leq l(s)$. The dictionary is $\mathcal{D}$. The length of a word is $l(w)$, and its i-th letter is $w[i]$. Given a low-level variable (i.e., cell) c, there are at most two slots $s_H(c)$ and $s_V(c)$ [2] that contain it. For the simplicity of the presentation, we assume that there are exactly two slots that contain a cell. The position of the cell c in the slot $s_t(c)$ is denoted as p_t^c, $t \in \{H, V\}$. For an instantiated high-level or low-level variable x, we denote its assigned value by $v(x)$.

3.1 High-Level Search Space

Searching for a solution is performed at the high level, where one move is to select both a slot and a word to be added to the grid. Selecting a slot–word pair uses a variant of the most constrained heuristic, a principle widely used in CSP applications: find the most constrained word slot s according to the $K(s)$ measure defined below and fill it with an word that introduces the least amount of new constraints and thus gives more options in continuing filling the grid.

Compared to a naive branching scheme where variables are instantiated in order (e.g., slots starting from the upper left corner and words in alphabetical order), the most constrained heuristic is a big win in the problem of crossword grid composition. Puzzles that are next to impossible to fill with the naive approach can become very easy. However, this enhancement alone reaches its limits quickly in many interesting puzzles. The performance can dramatically be improved with the hierarchical encoding.

Our measure of how constrained a slot s is considers both the number of matching words $N(s)$ according to the current constraints (see details below) and the length of the slot $l(s)$. Slots that have all cells filled with letters as a result of previous moves are not

[2] H and V stand for "horizontal" and "vertical" respectively.

considered in the following discussion, unless otherwise mentioned. In the definition below, a smaller $K(s)$ corresponds to a more constrained slot s:

$$\begin{aligned} K(s) = {} & 0, \text{if} N(s) = 0 \\ & 1, \text{if} N(s) = 1 \\ & 1 + \frac{N(s)}{l(s)^2}, \text{otherwise} \end{aligned}$$

For a given partially filled grid and a slot s, define $W^*(s)$ as the set of all words that are placed on slot s in at least one solution (i.e., correct completion) of that partial grid. Computing $W^*(s)$ is as hard as finding all solutions to a puzzle. $W^*(s)$ is approximated with $W(s)$, a superset computed according to the current knowledge about the grid at hand. $N(s) = |W(s)|$ is an upper bound of $N^*(s) = |W^*(s)|$. The condition $W^*(s) \subseteq W(s)$ preserves the completeness of the search. In this work we discuss several methods for computing $W(s)$. As shown later, the version of W computed with a given method is denoted by W_i, where i is a index.

Before describing how $W(s)$ is computed, we emphasise the key observation that smaller sets are more desirable as long as completeness is preserved. This becomes more obvious after we discuss its most important effects in search. Firstly, it determines what area of the grid to fill next, since a slot with the smallest $K(s)$ is tried first. Secondly, $N(s^+)$, where $s^+ = \arg\min_s K(s)$, represents the number of successors (branching factor) of the current node. See a formal result about the branching factor later. Thirdly, slots for which no words match the current constraints have $K(s) = 0$ and thus rank at the top of list, allowing to detect a deadlock before trying to fill other parts of the grid. Fourthly, when exactly one word fits in a slot, adding it to the grid as soon as possible can only help. This is similar to forced moves in games and unit propagation in SAT.

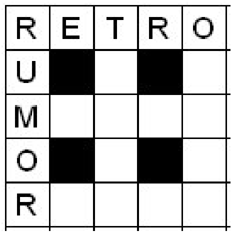

Fig. 1. Partially filled grid used as a running example.

Figure 1 shows a toy problem that we use as a running example. The slot $s_{H,i}$ corresponds to the i-th row, whereas the slot $s_{V,i}$ corresponds to the i-th column. Words RETRO and RUMOR have been added to the grid as shown in the picture. Assume that, besides the already added words, the dictionary contains the following entries: MACRO, MAGDA, MAGIC, MARTE, MASAI, MATRI, MEDIC, METRO, MOGUL, MOTOR, OARED, OCCUR, OPALS, OPERA, OPIUM, OPTIN, ORION, ORGAN, RADAR, RADIO, RARED, REBUS, ROBOT, ROMAN, ROTOR, TABBY, TABLA, TABLE, TABOR, TEMPO, TIGER, TORID, TREND. As shown in the experiments section, our system handles much larger puzzles. This simple example is used just for a better understanding of the method.

The remainder of this section focuses on how $N(s) = |W(s)|$ is computed using only the high-level encoding. This is roughly equivalent to what Ginsberg *et al.* call lookahead in their work [3]. It should be seen as the base model that we extend as shown in this work. We denote by $W_0(s)$ the approximations of $W^*(s)$ computed using only the high level. For a given slot s, $W_0(s)$ is dynamically updated each time when the most recently added word intersects s. Part of the cells of the slot s can contain letters that were added when instantiating slots that intersect s:

$$\begin{aligned}
&(\exists k) 0 \le k < l(s)\\
&(\exists i_1 \dots i_k) 1 \le i_1 < \dots < i_k \le l(s) :\\
&v(s[i_1]) = \alpha_{i_1} \wedge \dots \wedge v(s[i_k]) = \alpha_{i_k},\\
&\alpha_{i_j} \in \{A \dots Z\}
\end{aligned}$$

$W_0(s)$ contains all words in the dictionary that respect the constraints introduced by the already filled cells and have no constraints on the empty cells:

$$W_0(s) = \{w \in \mathcal{D} | l(s) = l(w) \wedge \forall m = 0 \dots k : w[i_m] = \alpha_{i_m}\}$$

Variable	Domain
Slot $s_{H,3}$	MACRO, MAGDA, MAGIC, MARTE, MASAI, MATRI, MEDIC, METRO, MOGUL, MOTOR
Slot $s_{H,5}$	RADAR, RADIO, RARED, REBUS, ROBOT, ROMAN, ROTOR,
Slot $s_{V,3}$	TABBY, TABLA, TABLE, TABOR, TEMPO, TIGER, TORID, TREND
Slot $s_{V,5}$	OARED, OCCUR, OPALS, OPERA, OPIUM, OPTIN, ORION, ORGAN

Table 1. Variable domains computed with the high-level encoding.

In the running example, the words RETRO and RUMOR reduce the domains of the slots intersected by each word. Table 1 contains the updated domains of each slot. The most constrained slot is $s_{H,5}$ (i.e., the one on the fifth row), since there are only seven words that start with an R. Hence this method selects $s_{H,5}$ to be filled next, and the branching factor is seven. Figure 2 shows a few successors of the current grid that are computed with the high-level encoding.

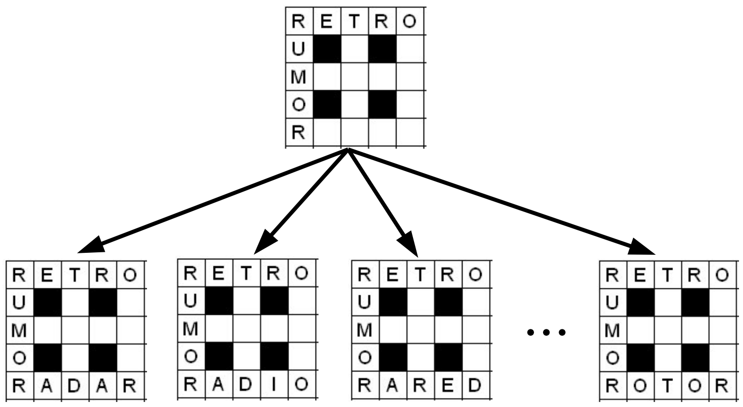

Fig. 2. Successors computed with the high-level encoding. Four out of seven successors are shown.

3.2 Low-Level Encoding and Channelling Constraints

As a low-level problem representation, each non blocked cell introduces a variable whose domain is the set of the alphabet characters $\mathcal{A} = \{A, \ldots, Z\}$. The low-level encoding is used to reduce the sets $W_0(s)$ and thus make the high-level search more informed, as discussed before. In this section we describe an iterative method for reducing $W_0(s)$. The approximation of $W^*(s)$ computed after i iterations is denoted by $W_i(s)$. As soon as a variable domain is reduced to the empty set, a deadlock has been detected and there is no need to move on to the next iteration. Otherwise, iterations are repeated for a number of times given as a parameter or until a fixpoint is reached.

Each iteration is a two-way propagation of channelling constraints between the two levels. First, a downwards propagation updates the domain of each cell based on the domains of the slots that contain it. Then, the updates in each cell domain are propagated back to the high level, further restricting the domains of word slots.

More formally, consider two high-level variables (slots) s_H and s_V that have a common uninstantiated low-level variable (empty cell) c. Assume c is at the intersection of the p^c_H-th position of the slot s_H and the p^c_V-th position of the slot s_V. Define $C^*(c)$ as the set of letters that appear on cell c in at least one correct completion of the partial grid at hand. For a cell c that is already instantiated with a letter α, $C^*(c) = C(c) = \{\alpha\}$. For a blank cell, $C^*(c)$ is approximated with a superset $C(c)$ computed as below. $C(c) = C_H(c) \cap C_V(c)$, where $C_t(c) = \{\alpha \in \mathcal{A} | \exists w \in W_{i-1}(s_t) : w[p^c_t] = \alpha\}$, $t \in \{\mathrm{H}, \mathrm{V}\}$. Said in simpler words, we compute what letters might be added to that cell by the words on the horizontal slot, compute a similar set of letters induced by the vertical set, and take the intersection of the two sets. See the running example below.

Given a slot s, upwards propagation can further reduce $W_{i-1}(s)$ by considering the sets $C(c)$ of all cells c that belong to s:

$$W_i(s) = \bigcap_{k=1..l(s)} \{w | l(s) = l(w) \wedge w[k] \in C(s[k])\}$$

In the running example, the blank cells at the intersection of two slots are $c_{3,3}$, $c_{3,5}$, $c_{5,3}$ and $c_{5,5}$, where the first number is the row in the grid and the second one is the column. For each of these, we compute a set of acceptable letters by looking at the words that fit into each of the two intersecting slots. For example, let us focus on the cell $c_{3,3}$, located at the intersection of $s_{H,3}$ and $s_{V,3}$. Consider $W_0(s_{H,3})$, the set of words that fit into the third row. These are all words that start with an M. All letters that appear on the third position of these words are C, D, G, R, S, T. Similarly, consider all words that fit into the third column (i.e., words that start with T). B, E, G, M, R are all letters that appear on the third position of these words. In effect, the intersection G, R is the set of acceptable letters for cell $c_{3,3}$. According to the previous notations, $C(c_{3,3}) = \{G, R\}$. Table 2 shows the updated domains for all low-level and high-level variables. Note that all slots are significantly more constrained than in Table 1. The most constrained slots are $s_{H,5}$ and $s_{V,3}$. Either one can be selected to be instantiated next. In Figure 3, slot $s_{V,3}$ is preferred to illustrate that the selected slot can change as compared to Figure 2. The branching factor reduces from seven to two.

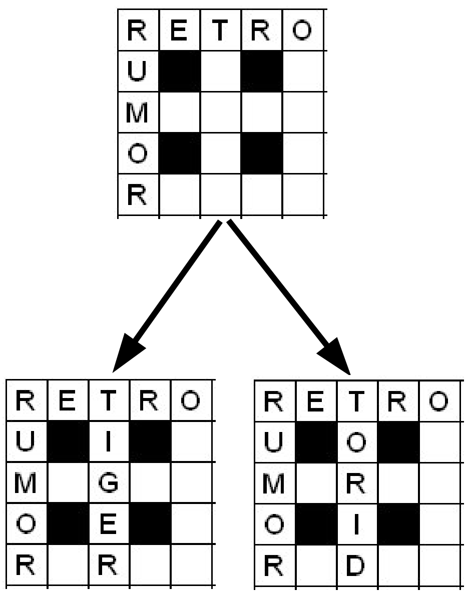

Fig. 3. Successors computed after one iteration.

Figure 4 shows the iterative procedure in pseudocode. Table 3 shows how the procedure is run on the example. Iteration 1 is the same as in Table 2. The following iterations keep reducing the domains of each variables. After iteration 4, a deadlock is discovered. The grid in Figure 1 is proven to be deadlocked without resorting to search.

Variable	Domain
Cell $c_{3,3}$	G, R
Cell $c_{3,5}$	A, C, E, I, R
Cell $c_{5,3}$	D, R
Cell $c_{5,5}$	D, N, R, S
Slot $s_{H,3}$	MAGDA, MAGIC, MARTE
Slot $s_{H,5}$	RADAR, RARED
Slot $s_{V,3}$	TIGER, TORID
Slot $s_{V,5}$	OARED, OCCUR, OPALS, ORION

Table 2. Variable domains computed after one iteration.

```
1: $i \leftarrow 0$
2: **repeat**
3:     $i \leftarrow i + 1$
4:     {downwards propagation:}
5:     **for** all empty cells $c$ **do**
6:         $C_H(c) \leftarrow \{\alpha \in \mathcal{A} | \exists w \in W_{i-1}(s_H(c)) : w[p^c_H] = \alpha\}$
7:         $C_V(c) \leftarrow \{\alpha \in \mathcal{A} | \exists w \in W_{i-1}(s_V(c)) : w[p^c_V] = \alpha\}$
8:         $C(c) \leftarrow C_H(c) \cap C_V(c)$
9:     **end for**
10:    {upwards propagation:}
11:    **for** all slots $s$ that contain at least one empty cell **do**
12:        $W_i(s) \leftarrow \{w \in \mathcal{D} | l(w) = l(s) \wedge \forall k(w[k] \in C(s[k]))\}$
13:    **end for**
14: **until** no change occurs or $i =$ max_iterations or a deadlock is found
```

Fig. 4. Iteratively propagating constraints between the hierarchical levels. A deadlock is found when the domain of a variable becomes empty.

To have the results below hold for both uninstantiated and instantiated slots, we extend the definition of W to the trivial case of a slot s already filled with a word w: $W_i(s) = W^*(s) = \{w\}, \forall i \geq 0$.

Theorem 1. *For a given partially filled grid, a slot s and* $0 \leq i < j$, $W_i(s) \supseteq W_j(s) \supseteq W^*(s)$.

Corollary 1. *For a given partially filled grid and* $0 \leq i < j$, *if* W_i *detects a deadlock then* W_j *detects that deadlock too.*

Corollary 2. *For a given partially filled grid and* $0 \leq i < j$, *if* W_i *detects a forced move then* W_j *detects either a forced move or a deadlock.*

Since the definition of $K(s)$ includes the length of a slot $l(s)$, it is possible that the branching factor of a given partially filled grid increases as one more propagation iteration is performed. Although doing more propagation cannot increase the domain sizes of the slot variables, it can cause the most constrained variable to become one

Variable	Iteration 2	Iteration 3	Iteration 4
Cell $c_{3,3}$	G, R	G	G
Cell $c_{3,5}$	A, C	C	C
Cell $c_{5,3}$	D, R	D, R	R
Cell $c_{5,5}$	D, R	R	$\emptyset$
Slot $s_{H,3}$	MAGDA, MAGIC	MAGIC	
Slot $s_{H,5}$	RADAR, RARED	RADAR	
Slot $s_{V,3}$	TIGER, TORID	TIGER	
Slot $s_{V,5}$	OCCUR	OCCUR	

Table 3. Variable domains computed iteratively. The first iteration is shown in Table 2.

corresponding to a longer slot, and so the branching factor can increase. The following corollary shows a condition that is sufficient to guarantee that the branching factor does not increase.

Corollary 3. *Consider a fixed partially filled grid and its corresponding search node n. For a number of propagation iterations i, consider s_i the selected slot for that grid and $b_i = |W_i(s_i)|$ the branching factor of n. If $i < j$ and $l(s_i) \geq l(s_j)$ then $b_i \geq b_j$.*

Proof. For simplicity, we skip the cases when $K(s) = 0$ or $K(s) = 1$. Define $K_i(s) = 1+\frac{|W_i(s)|}{l(s)^2}$, the version of $K(s)$ computed with $W_i(s)$ in use. Since $s_j = \arg\min_s K_j(s)$,

$$1 + \frac{|W_j(s_j)|}{l(s_j)^2} = K_j(s_j) \leq K_j(s_i) = 1 + \frac{|W_j(s_i)|}{l(s_i)^2} \Leftrightarrow$$
$$\frac{|W_j(s_j)|}{l(s_j)^2} \leq \frac{|W_j(s_i)|}{l(s_i)^2} \Leftrightarrow$$
$$|W_j(s_j)| \times l(s_i)^2 \leq |W_j(s_i)| \times l(s_j)^2.$$

Since $l(s_j) \leq l(s_i)$, $|W_j(s_j)| \leq |W_j(s_i)|$. Since $i < j$, $|W_j(s_i)| \leq |W_i(s_i)|$, according to the theorem above. From the last two inequalities, we get that $b_j = |W_j(s_j)| \leq |W_i(s_i)| = b_i$.

3.3 A More Generally Applicable Hierarchical Model

In this section we discuss how the lessons learned from this work can help to design a hierarchical CSP model applicable to a larger range of CSP problems, not just crossword composition. Specifically, we focus on automatically building a hierarchical representation starting from a classical CSP encoding. The resulting problem can be solved as shown in the previous sections. In converting a problem representation from classical into hierarchical CSP, the original encoding becomes the low level of the new hierarchy. Several low-level variables can then be combined into a high-level variable, just as several cells compose a word slot. A value of a high-level variable is a tuple with values of the contained low-level variables. The domain of each high-level variable is the set of tuples that respect all constraints between the contained variables but ignores all other constraints.

In crosswords, the domain of each high-level variable is already given as a dictionary. In the more general model, the high-level variable domains have to be computed beforehand. In terms of worst-case complexity, computing each variable domain is exponentially easier than solving the original problem. In domains where some constraints are fixed and some other vary with the problem instance, the high-level variable domains could be reused from one instance to another, similarly to reusing the English dictionary in crosswords.

This hierarchical model can be seen as a form of problem decomposition. Each high-level variable is a subproblem. All subproblems are solved in advance and all their solutions are cached as macro-actions. When the global problem is solved, a combination of macro-actions is sought that respects all constraints between subproblems.

4 Search Strategy

The problem space is explored in a depth-first manner. This choice is supported by the bounded depth of a search tree (it never exceeds the total number of word slots) and by the small memory requirements. When expanding a node, a slot s is selected as shown in the previous section. Each word in $W_i(s)$ (where $W_i(s)$ is a superset of $W^*(s)$ computed with one of the methods described before) generates a successor. The successors of a node are sorted according to the most constrained heuristic. Each node is assigned a score computed as $\prod_{s \in S} N(s)$, where S is the set of all slots yet to be instantiated and $N(s) = |W_i(s)|$. Nodes with a higher score are ranked first.

When a node is proven deadlocked (because either the branching factor is zero or all successors were explored and no solution was found), a deadlock pattern is extracted from the partially filled grid by preserving the values of some instantiated cells and ignoring the rest of the grid. A deadlock pattern is a partial assignment of the low-level encoding. All grids that contain that assignment can be pruned from the search space.

To compute such a pattern, a deadlock area is built starting from the slot that has no valid words and thus has caused the grid to be labelled as deadlocked. Empty of partially filled slots that intersect the deadlock area are used to grow it until a fixpoint is reached. Even if fully instantiated slots are skipped, part of their cells might be added to the deadlock area by intersecting partially instantiated slots.

All instantiated cells in a deadlock area are a superset of a deadlock pattern. The current program version stores such supersets in a simple deadlock database that contains the most recent k records, where k is a parameter. (In experiments, k is set to 1,000.) It might be useful to process a deadlock superset and extract smaller and thus more general patterns. As a heuristic rule, cells in the proximity of the deadlock slot should be tried first. Note that the deadlock handling mechanism exploits the dual encoding of the problem. Deadlocks are used in the high-level search space but are stored as partial assignments at the low level.

The search algorithm implements a *solution sampling* idea. Its presentation is important not only from a practical perspective, but also for a better understanding of the search strategy and the experiments. For most problems in real life, finding one (good) solution is sufficient. In crosswords, many solutions need to be enumerated, such that a publisher can release one new puzzle every day. The order in which solutions are enumerated is important. If a naive strategy is employed, one solution will be followed by countless very similar variations, which are not interesting to users.

A better approach is to output a small number of similar puzzles (such that a human expert can select the best one) and then explore a new part of the search space, where solutions are significantly different. In our current implementation, reaching a solution is followed by a backjumping directly to the root, whose next successor is explored. Extensions such as enumerating a few similar solutions or backjumping over a smaller, user-specified number of steps are easy to add.

5 Experimental Evaluation

The ideas presented in this paper were implemented in C++. The resulting system is called COMBUS. A first experiment compares COMBUS against previous state-of-the-art results [1]. In a second experiment, we evaluate the impact of using two viewpoints as compared to a single viewpoint.

We use the same problem set as in [1]. The data contain ten grids for each of the following sizes: 5x5, 15x15, 19x19, 21x21 and 23x23. There are two dictionaries, containing 45,000 and 220,000 words respectively. Each combination of a grid and a dictionary creates a problem instance, obtaining a set of 100 problems. In this experiment, the program stops after finding one solution to the problem instance at hand. COMBUS is set to perform five iterations of channelling constraint propagation. Beacham *et al.* have run the experiments on a 300MHz machine with a time limit of 10 hours per problem. Our machine is six times faster, so we limit the time to 6,000 seconds per problem. Beacham *et al.* evaluate the performance of several combinations of algorithms, heuristics and encodings. The top performer in that work solves 92 problems. COMBUS solves 95 problems.

Dictionary	Grid Size	Solved	Nodes	Time (seconds)
small	15x15	9	54,654	545
large	15x15	10	703	789
small	19x19	10	19,070	480
large	19x19	10	1,195	1,473
small	21x21	8	181,550	2,836
large	21x21	10	2,018	2,366
small	23x23	8	122,814	2,119
large	23x23	10	2,094	2,659

Table 4. Summary of results with a 6,000 second limit. Each row corresponds to a set of ten grids. Columns 4 and 5 show the total effort to solve a number of problems indicated in the third column. For each problem, only one solution is sought.

The results are summarized in Table 5. The smaller dictionary produces harder problems. A similar finding is mentioned in [1]. The explanation seems to be that, with a big dictionary, the deadlocks are less frequent and the "density" of goal states in the problem space is larger. With a large dictionary, all problems are solved with few nodes expanded. Often, solutions are found without backtracking. Note that a larger dictionary increases the cost to process one node. It can be as high as one node per second, but there is significant room to improve it with a better implementation.

In the problem set summarized on the first row of Table 5, two problems dominate the total effort, accounting for 93% of the total node expansions. All remaining solved problems require less than 2,000 nodes each. Three instances are solved after expanding less than 100 nodes each.

On the third row, one problem takes 90% of the total number of expanded nodes. Search in all other problems succeeds within less than 500 nodes. On the fifth row, there is one hard instance (besides the two unsolved ones), which is responsible for 98% of the total search effort measured as expanded nodes. The other problems require at most 1,030 nodes each. On the second last row, one problem requires over 80,000 nodes, one about 33,500 nodes, and one almost 8,000 nodes. The remaining ones are solved within less than 550 node expansions in each case. Even rows, which correspond to the large dictionary, have low values for the number of expanded nodes and do not require a further discussion.

In the second experiment, the hierarchical architecture is compared against the standard model with one viewpoint containing slot variables. Three system configurations are run. W_0 corresponds to the basic model. W_1 is the hierarchical system with one iteration for propagating the channelling constraints. W_5 is the version with five iterations. For each problem, the program attempts to find 5 solutions to each problem (recall the solution sampling strategy). For faster results, the dataset is reduced to the 15x15 and 19x19 grids and the time per problem is limited to 30 minutes.

		W_0				W_1				W_5			
Dict.	Size	S	NS	Nodes	Time	S	NS	Nodes	Time	S	NS	Nodes	Time
small	15x15	6	21	135,593	1,243	8	35	385,086	2,653	9	39	432,790	3,561
large	15x15	10	46	48,521	1,911	10	50	4,492	3,249	10	50	3,592	3,868
small	19x19	10	33	326,217	3,341	10	34	329,545	3,754	10	40	52,081	1,570
large	19x19	10	46	11,167	4,623	10	50	6,346	6,309	10	50	5,929	6,992

Table 5. Standard encoding vs hierarchical encoding.

In Table 5, S is the number of problems for which at least one solution is found. NS is the total number of solutions found for the ten problems in the corresponding set. The node and time data include the total effort to find all the solutions reported in the NS column. The table does not report the effort spent in unsuccessful searches. Hence it is normal to have a larger number nodes or seconds for a program that finds more solutions.

The program versions corresponding to W_1 and W_5 find more solutions than W_0, showing that the hierarchical model improves the solver. As before, the data shows that the instances with a small dictionary are harder. In hard problems, W_5 solves more problems than W_1, indicating that more iterations are desirable. In the easy problems corresponding to a large dictionary a larger number of iterations still reduces the number of expanded nodes. However, the increased cost per node results in a degradation of the overall performace as compared to W_1. For easy instances, a small number of propagation iterations seems to be a good trade-off to balance the number of expanded nodes and the cost to process one node.

6 Conclusion

This paper has introduced an approach to automatic crossword grid composition that represents a problem on two hierarchical levels. The high level defines a variable for each word slot, whereas individual cells correspond to low-level variables. Search is performed at the high level, instantiating entire slots rather than single cells. The low level is useful to reduced the search space via a collection of channelling constraints. In experiments, the hierarchical model solves more puzzles compared to a classical encoding and to previous results reported in the literature.

A promising idea for the future is dynamic problem decomposition. Depending on the arrangement of blocked cells, a problem might quickly become decomposable into independent parts after instantiating relatively few slots. A second idea would be investigate the impact of backjumping and dynamic variable ordering on the system performance. It would be interesting to study empirically how the difficulty of a puzzle varies with parameters such as dictionary size, grid size, and number of blocked cells. We are interested in generating thematic puzzles, where part of the words belong to a given theme (e.g., movie titles). The challenge would be to maximize the number of thematic words. We plan to generalize the hierarchical model presented in this paper into a method applicable to any CSP problem, not only crossword grid composition.

7 Acknowledgment

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

We are grateful to Markus Enzenberger for developing a graphical user interface for the COMBUS engine, and to Martin Müller and Jonathan Schaeffer for their feedback at earlier stages of this work. We thank the anonymous reviewers for their comments and suggestions, and Peter van Beek for providing us with test data to be used in the experiments.

References

1. A. Beacham, X. Chen, J. Sillito, and P. van Beek. Constraint Programming Lessons Learned from Crossword Puzzles. *Lecture Notes in Computer Science*, 2056:78–87, 2001.

2. M. Ernandes, G. Angelini, and M. Gori. WebCrow: A WEB-based system for CROssWord Solving. In *Twentieth National Conference on Artificial Intelligence AAAI-05*, pages 1412–1417, 2005.
3. M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search Lessons Learned from Crossword Puzzles. In *Eighth National Conference on Artificial Intelligence*, pages 210–215, 1990.
4. B. Hnich, S. Prestwich, and E. Selensky. Modeling the Covering Test Problem. 2004.
5. M. L. Littman, G. A. Keim, and N. Shazeer. A Probabilistic Approach to Solving Crossword Puzzles. *Artificial Intelligence*, 134(1-2):23–55, 2002.
6. A. K. Mackworth, J. A. Mulder, and W. S. Havens. Hierarchical Arc Consistency: Exploiting Structured Domains in Constraint Satisfaction Problems. *Comuptational Intelligence*, (1):118–126, 1985.
7. L. J. Mazlack. Computer Construction of Crossword Puzzles Using Precedence Relationships. *Artificial Intelligence*, (7):1 – 19, 1976.
8. G. Meehan and P. Gray. Constructing Crossword Grids: Use of Heuristics vs Constraints, 1997. `citeseer.ist.psu.edu/433222.html`.
9. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, chapter 11 Modelling, pages 377–406. 2006.