Deductive Search for Logic Puzzles

Cameron Browne
Imperial College London
South Kensington, UK
camb@doc.ic.ac.uk

Abstract—Deductive search (DS) is a breadth-first, depth-limited propagation scheme for the constraint-based solution of deduction puzzles, using simple logic operations found in standard constraint satisfaction solvers. It attempts to emulate the processing limits experienced by human solvers, and, to some extent, the process by which they solve such problems. Any solution deduced by DS is guaranteed to be correct and unique. Further, it provides an estimate of the deducibility of a given problem for human solvers and offers new ways of understanding deduction puzzles. Its performance is tested on a number of problem domains including Japanese logic puzzles, a traditional logic puzzle, and a geometric placement puzzle.

I. INTRODUCTION

Sudoku is currently rivalling the crossword as the world’s most popular “pencil and paper puzzle”, largely because it is language-neutral and can be solved using pure logic without the need for culture-specific knowledge [1]. It is representative of a family of logic puzzles, popularised by Japanese publisher Nikoli and hence called called Japanese logic puzzles [2], that are characterised by the following properties:

1) Single player.
2) Simple rules.
3) Single unique solution.
4) Can be solved by deduction (not guesswork).
5) Culture-independent and language-neutral.

For example, Figure 1 shows Slitherlink, a typical Japanese logic puzzle, in which the player must draw edges on a grid to form a single non-self-intersecting closed path, such that the number of edges around each numbered cell equals that number. We will refer to such puzzles as deduction puzzles.

![Fig. 1. A Slitherlink challenge (left) and its solution (right).](image)

There exist various techniques for solving deduction puzzles, which can be divided into two broad categories:

1. Heuristic Approaches: These solvers use heuristics, rules, strategies or patterns specific to the given domain, usually modelling approaches that players (i.e. human solvers) would apply. Such solvers are tied closely to their given domain, and can require expert knowledge about the domain to operate successfully. Difficulty ratings derived from heuristic solvers can be unreliable, as they may not implement all strategies, and strategies that one player finds difficult could be easy for another. For example, in a comparison of 375 difficult Sudoku puzzles, there was no agreement in the five examples that three popular solvers (Q1/2, SE and SUEXRAT) each rated most difficult [3].

2. Mathematical Models: Puzzles may be abstracted to mathematical models and solved using standard constraint satisfaction problem (CSP) [4], Boolean satisfiability (SAT) [5], binary decision diagram (BDD) [6] or other optimisation techniques. These approaches are general, well studied and understood, and typically efficient at finding solutions. However, they do not necessarily represent the player’s understanding of a given puzzle. Firstly, the mapping of the problem to a mathematical model can rephrase it in terms that a human might not recognise (e.g. 9×9 Sudoku maps to 729 binary SAT variables [5]). Secondly, these approaches tend to be recursive in nature to arbitrary search depths, and not subject to the same processing limitations experienced by players.

A. Motivation

Recursive structures are hard for humans to model mentally [7]. Evidence from the field of psycholinguistics suggests a mental limit of two levels of embedding (2-LoE) in natural language processing [8]. Such sentences, constructed with two recursive levels of embedding of grammatical rules, are occasionally found in written form but almost never in natural discourse, and there have been no known cases of natural 3-LoE sentences in over a century. Each level of recursion requires mental storage of embedding points for future backtracking, which quickly exhausts short-term memory [7].

We posit that a similar limit applies to mental recursion employed by players when solving deduction puzzles, as they must mentally store embedding points (and any resulting consequents) of lookahead moves. Further, we believe that this limit should be respected by any automated solver that aims to emulate the human puzzle-solving process with any accuracy.

With this in mind, we propose a new approach for solving deduction puzzles called deductive search (DS). This is a breadth-first, depth-limited, constraint-based approach based on known techniques, but implemented to model puzzles as directly as possible and use simple logic operations that a player would typically employ under similar computational limitations. The aim of DS is to determine the deducibility of a given puzzle challenge for humans. The following sections describe the algorithm and its performance on a variety of deduction puzzles.

II. DEDUCTIVE SEARCH

Deductive search (DS) is an iterative constraint-based approach for solving puzzles using simple logic operations in a
A. Model

Each domain is modelled as a CSP involving a set of variables \( X = \{X_1, X_2, \ldots, X_n\} \) with integer values in their respective domains \( D = \{d_1, d_2, \ldots, d_n\} \), where \( D_{X_i} \) is the domain of possible values for \( X_i \). Each variable represents a decision to be made and must have exactly one correct value, otherwise \( X \) does not constitute a deduction puzzle. The instantiation of a variable \( X_i \) to value \( v \) is denoted \( X_i \cdot v \) and the elimination of value \( v \) from \( X_i \) is denoted \( X_i \setminus v \).

The current state \( S \) represents the set of values \( X \) at a given point. The initial state \( S_0 \) is called the challenge. \( C = \{c_1, c_2, \ldots, c_m\} \) is a set of constraints that every state \( S \) must satisfy, where the constraint \( c_j \) is defined on a subset of variables \( \{X_{i_1}, X_{i_2}, \ldots, X_{i_k}\} \subset X \), where \( a_j \) is the arity of constraint \( c_j \). A variable \( X_{i_j} \) is arc-consistent with another variable \( X_{i_j}' \) if each value \( v \in X_{i_j}^* \) is consistent with at least one value \( w \in X_{i_j}^* \).

The subset of unresolved variables for a given state \( S \) is denoted \( S^* \), and the subset of remaining values available to a given variable \( X_i \) is denoted \( X_i^* \). \( |X_i|, |X_i^*| \) and \( |S^*| \) denote the cardinality of the respective sets. A state \( S \) is solved when all variables are resolved to their correct value (i.e. \( S^* = \emptyset \)) and the instantiation of \( X \) is consistent with \( C \).

B. Operation

Given a state \( X \), variable updates (instantiations and eliminations) are propagated iteratively in three ways.

1. Simplification. For each update to a variable \( X_{i} \), perform updates to other variables in each constraint \( C_j : X_{i} \in C_j \) required to maintain arc-consistency.

   This operation simplifies variables to maintain arc-consistency within the domain of each constraint \( C_j \) of which \( X_i \) is a member. The exact simplification performed in each case depends on the type of constraint. For example, the instantiation of a cell to value \( v \) in Sudoku means that \( v \) can be eliminated from all other cells in its row, column and sub-grid.

2. Shaving. For each unresolved variable \( X_{i} \), any available value \( v \) whose instantiation would create a contradiction \( \bot \) (following simplification) is eliminated:

   \[
   (\exists v \in X_{i}^* : X_{i} \cdot v \Rightarrow \bot) \Rightarrow X_{i} \leftarrow X_{i} \setminus v
   \]  

   Invalid instantiations that would result in a contradiction are shaved from their variables [9]. For example, consider the decision \( X_i \) (dotted) in the 3×2 Slitherlink example shown in Figure 2. There are two values available to \( X_i \), which yields two possible hypotheses:

   \[
   H_0 : X_i = 0 \text{ (no edge)} \\
   H_1 : X_i = 1 \text{ (edge)}
   \]

   \( H_0 \) leads to a contradiction (following simplification) as the hint with value 3 is violated (top row). Hence, \( H_0 \) is rejected, whereas the shaving and agreement steps are standard logic operations found in SAT and CSP solvers that constitute the

![Fig. 2. Shaving of value 0 from \( X_i \) due to contradictory hypothesis \( H_0 \).](image)

0 is eliminated from \( X_i \), and \( X_i \) resolves to the only remaining value (1, right).

Shaving, in this context, is the integer-based counterpart of the failed literal rule used to propagate binary SAT variables in the standard DPLL algorithm [10]. The opposite operation (instantiating values whose elimination would cause a contradiction) was found to have little benefit in practice for DS, so it is not implemented here.

3. Agreement. For each unresolved variable \( X_i \), if every potential instantiation \( v \) in \( X_j \) (following simplification), then \( X_j \) is instantiated to \( v \):

   \[
   (\forall v \in X_{j}^* : X_{j} \cdot v \Rightarrow X_j \cdot w) \Rightarrow X_j \leftarrow X_j \cdot w
   \]  

Conversely, if no potential instantiation of any \( v \) in \( X_i \) results in the instantiation of any \( w \) in \( X_j \) (following simplification), then \( w \) is eliminated from \( X_j \):

   \[
   (\exists v \in X_{i}^* : X_{i} \cdot v \Rightarrow X_j \cdot w) \Rightarrow X_j \leftarrow X_j \setminus w
   \]

Values are therefore updated if that update would occur as a result of every possible instantiation of some other variable. For example, Figure 3 shows this process applied to a variable \( X_i \) (dotted, left) in another 3×2 Slitherlink example.

![Fig. 3. Instantiation of other values due to agreement with \( X_i \).](image)

Again, the two possible hypotheses for \( X_i \) are \( X_i = 0 \) (no edge) and \( X_i = 1 \) (edge). However, this time the focus is on the simplifications resulting from \( X_i \) rather than \( X_i \) itself (Figure 3, middle). Since both hypotheses result in the same three variables being simplified to 1 (edge), these variables can safely be instantiated to this value (right), even though no conclusions can be drawn about \( X_i \) itself.

Simplification depends on the constraints being modelled, whereas the shaving and agreement steps are standard logic operations found in SAT and CSP solvers that constitute the
“deductive” part of the search. They are equivalent to the steps used by Herting to generate local patterns for Slitherlink [11], but are here generalised to arbitrary domains and used to propagate the broader search.

C. Algorithm

Listing 1 shows how these three standard operations are combined to produce the DS algorithm. Given a state S to solve, the search begins with a straight simplification pass (0-LoE) over all unresolved variables Xi ∈ S* to perform any obvious variable updates. DS then enters a loop that repeatedly performs the deduction steps, until S is solved or no more updates are found:

1) Repeatedly apply 1-LoE deduction passes until either no more updates are made or S is solved.
2) If not solved, apply a 2-LoE deduction pass.

The STATUS(S) function returns the current status of S, which will remain UNSOLVED (∅) until either a solution is deduced or a contradiction is proven. ∆S describes the subset of variables in S updated since the last check.

Each deduction pass applies the shaving and agreement steps to each unresolved variable Xi ∈ S*, to the specified depth. Both steps are performed in the same pass for efficiency. If an attempted instantiation Xi ∩ v and its simplifications do not lead to a contradiction, then the search recurses to the next depth (if depth > 1) and agreeing simplifications are accumulated in the on and off variables. Those simplifications common to all instantiations are then instantiated in S and those absent from all are eliminated from S.

Lines 28 and 29 perform an early escape if any deduction occurs at 2-LoE or deeper, in which case the search will revert back to 1-LoE, to minimise the recursive depth involved in each pass. A 3-LoE DS is achieved by repeating lines 7 and 8 back to 1-LoE, to minimise the recursive depth involved in each pass. This algorithm is an improvement on a previous version that did not include the agreement step [12].

D. Search Result

The search continues until S is resolved, a contradiction is proven, or the depth limit (in this case 2) applies. The search status of S at any point will therefore be one of:

1) UNSOLVED (∅): No solution proven or disproven yet.
2) SOLVED: A solution has been deduced.
3) CONTRADICTION: No possible solution exists.
4) NONDEDUCIBLE: No solution can be deduced with the current constraints and search depth.

A CONTRADICTION occurs when any constraint is violated, in which case S has no valid solution. S is deemed NONDEDUCIBLE if its status remains ∅ at the end of the search and |S*| < 2, as all possible combinations of remaining variables will have been tried without either solution or contradiction.

Non-deducible cases occur if either S is ambiguous and has multiple solutions, the constraints are insufficient, or search depth 2 is not sufficient. For example, the Slitherlink challenge shown in Figure 3 is non-deducible as it has multiple solutions (Figure 4). Any of these solutions may be found through guesswork, but not through deduction.

Algorithm 1 Deductive Search

```
1: function DS(S)
2:   S ← SIMPLIFY(S) /* 0-LoE */
3:   do
4:      S ← DEDUCE(S, 1) /* 1-LoE */
5:      while ∆S ≠ ∅ and STATUS(S) = ∅
6:     if STATUS(S) = ∅
7:        S ← DEDUCE(S, 2) /* 2-LoE */
8:     while ∆S ≠ ∅ and STATUS(S) = ∅
9:   end function
10: function DEDUCE(S, depth)
11:   for each Xi ∈ S* do
12:      on ← ∅
13:      off ← ∅
14:   for each v ∈ X_i do
15:      S' ← SIMPLIFY(X_i ∩ v)
16:      if STATUS(S') = ∅
17:         S ← SIMPLIFY(X_i ∩ v) /* shave */
18:      else
19:         if depth > 1
20:            S' ← DEDUCE(S', depth-1)
21:            on ← on ∩ S_i^*
22:            off ← off ∩ ¬S_i^*
23:      if on ≠ ∅
24:         S ← SIMPLIFY(S ∩ on) /* agree */
25:      if ¬off ≠ ∅
26:         S ← SIMPLIFY(S ∩ ¬off) /* agree */
27:   return
28: end function
```

Fig. 4. A non-deducible (ambiguous) case with multiple solutions.

E. Interface

The following functions are implemented for each domain:

1) START(S): Defines X, C and S0 for each challenge.
2) STATUS(S): Returns the status of S.
3) SIMPLIFY(S): Simplifies S according to constraints.

These functions encode the domain’s rules and provide the interface between the domain-specific constraints and the domain-independent deduction steps. Figure 5 shows the relationship between the various components. Each simplification only applies to those constraints relevant to the most recently updated variables ∆S.

The following constraints are implemented:

1A 2-LoE shaving is equivalent to the binary failed literary rule [5].
1) **ALL_DIFFERENT**: All \( X_i \in C_j \) have different values.

2) **EVEN**: All \( X_i \in C_j \) have even values.

3) **LESS_THAN** \((x)\): \( \sum_{i=1}^{n} X_{i} < x \).

4) **SUM_EQUALS** \((x)\): \( \sum_{i=1}^{n} X_{i} = x \).

5) **DISJOINT_SUM** \((x)\): \( \sum_{i=1}^{n} X_{i} = x \) (without repetition).

6) **PRODUCT_EQUALS** \((x)\): \( \prod_{i=1}^{n} X_{i} = x \).

7) **CONNECTED** \((type)\): Non-zero edges/vertices/cells are connected in graph \( G \).

8) **COVERS** \((type)\): Non-zero edges/vertices/cells cover \( G \).

9) **RELATION** \((i, v, j, w)\): \( X_i = v \iff X_j = w \).

10) **NO_COLLISION**: Bit sets for each \( X_i \) do not intersect.

### F. Deducibility

A challenge is described as being **deducible at depth** \( n \) if a \( n\text{-LoE} \) DS search produces a **SOLVED** result. If a solution is deduced, then that solution is guaranteed to be unique. The algorithm will not recognise solutions encountered during the search unless they are achieved through deduction.

Simonis deems a CSP problem to be deducible if it is “search free” and can be solved just by applying the constraints [4]. This definition also holds here, when one considers that DS is a propagation scheme for constraints; lookahead is applied to deduce information about the current state, not to find solutions directly.

The search depth constitutes a **deduction horizon** beyond which we cannot make any assumptions about the deducibility of a challenge. For example, Figure 6 shows a 3×4 Slitherlink challenge that is not solved by a 1-LoE search but is solved by a 2-LoE search.

Fig. 6. Not deducible at 1-LoE (middle) but deducible at 2-LoE (right).

**G. Difficulty**

A measure of the difficulty of solving state \( S \) is given as:

\[
diff(S) = S_0 + 4 \times (S_1 + V_1 + A_1) + 9 \times (S_2 + V_2 + A_2) + \sum_{i=1}^{n} D_i
\]

where \( S_n, V_n \) and \( A_n \) denote the number of updates due to simplification, shaving and agreement, respectively, at \( n\text{-LoE} \). The number of updates at each depth \( n \) is multiplied by a penalty factor \((n + 1)^2\) to reflect the increasing difficulty of each level of embedding, normalised by the total number of possible values involved.

DS is similar in principle to the **Ariadne’s Thread** strategy used by Sudoku players [13], in which combinations of unresolved variables are tested in order to detect contradictions. The name refers to the fact that the hypotheses, simplifications and embedding points projected by players constitute a mental “thread” that they must remember in order to backtrack to the originating state. This is recognised as a difficult strategy and is typically applied only as a last resort when all other strategies fail, so difficulty ratings produced by DS will probably tend towards the upper bound of actual difficulty.

### III. Experiments

The following Sections describe the algorithm’s application to a variety of deduction puzzles.

**A. Sudoku**

Sudoku is a Japanese logic puzzle³ in which the player must fill a square \( N \times N \) grid with the numbers 1 to \( N \), such that no number occurs twice in any row, column or sub-grid (where \( N = o^2 \)). The standard size is order \( o=3\) and starts with around 21–24 hints. Figure 7 shows an example 9×9 challenge from [14].

Fig. 7. The “World’s Hardest Sudoku” [14] and its solution.

**Constraints**: The **START** \((D)\) function creates a variable \( X_i \) for each grid cell with domain \( \{1,\ldots,N\} \), and instantiates hint cells to their known values. An **ALL DIFFERENT** constraint is created for each row, column and \( o \times o \) sub-grid.

**Results**: Table 1 shows the results of 2-LoE DS applied to a selection of Sudoku challenges. The \( |X|, |C| \) and \( |H| \) columns show the number of variables, constraints and hints, respectively. The \( s \) column shows the execution time in seconds, on a single thread of a Macbook laptop with \( i5 \) processor. The \( S_n, V_n \) and \( A_n \) columns show the number of instantiations.

---

³Despite being invented in the USA.
will cross its boundary an even number of times. For example, Figure 8 shows a row with an odd number of edges, hence the last remaining variable (dotted) must be 1. A CONNECTED(edges) constraint is created to ensure that all edges remain (potentially) connected, to ensure a single closed path.

A further constraint was hard-coded in the Slitherlink domain to facilitate solution. The ONE_OF constraint detects edge pairs around a vertex that must share exactly one edge, and propagates this knowledge along diagonal runs of hint cells with value 2 (Figure 9). This could have been implemented as a more general dynamic constraint, but was found to be more effective as a hard-coded constraint for this particular purpose. The inclusion of such “strategic” constraints specifically to facilitate solution is discussed further in Section IV.

B. Slitherlink

Slitherlink is a Japanese logic puzzle from Nikoli [17] in which the player must place edges between vertices in a square grid $G$ to form a single closed non-self-intersecting path, such that the number of edges around each hint cell equals that hint’s value. Figure 1 shows a typical challenge and its solution.

Constraints: The START($D$) function creates a variable $X_i$ for each adjacent vertex pair in $G$, with domain $\{0, 1\}$ indicating whether an edge joins them or not. A SUM_EQUALS($h$) constraint is created for each hint cell $h$. An EVEN and a LESS_THAN(3) constraint are created for each vertex in $G$, as any closed path must involve exactly 0 or 2 edges at each vertex. An EVEN constraint is created for each row and column below the Jordan Curve Theorem, as any simple (i.e. closed and non-self-intersecting) curve has an inside and an outside and any cross-section completely through it.

DS performs well for Slitherlink compared to existing methods, especially for more difficult challenges. For example, Hurlimann’s MIP constraint-based solver method [19] took 9 minutes to solve Janko challenge #100 while DS took less than 15 seconds. Further, Hurlimann’s method failed to solve

---

4. Slitherlink therefore maps neatly to a binary decision problem.
the Janko challenge #192 after an hour of computation, while DS solved it in under 4 seconds. In terms of performance relative to human players, Nikoli estimate that Slitherlink 21 challenge #96 will take a beginner approximately 98 minutes and an expert approximately 22 minutes to solve. DS solved this challenge in less that 4 seconds.

The $diff$ ratings produced by DS are reasonably consistent with each publisher’s ranking in order of difficulty. The Times challenge #75 appears to pack the most punch for its size, requiring the highest ratio of 2-LoE processing of any of the challenges tested.

Note that the $diff$ ratings for Slitherlink tend to be higher than those for Sudoku. This may be because elimination is more effective in a binary domain (such as Slitherlink) in which every elimination implies an instantiation, whereas multiple eliminations are typically required to instantiate non-binary variables; hence, relatively more of the processing effort goes on “behind the scenes” in the Sudoku domain. In any case, the $diff$ ratings produced by DS are more meaningful within each domain rather than between domains.

C. Hashiwokakero

Hashiwokakero is a Japanese logic puzzle from Nikoli [20], [21] in which the player must join hints in a square grid $G$ to form a single connected set, using only horizontal and vertical edges. The cardinality of each vertex must equal its hint value, edges cannot cross, and no more than two edges can connect any pair of vertices. Figure 10 shows an example from [21].

![Hashiwokakero challenge #15 from [2] and its solution.](image)

**Fig. 10.** Hashiwokakero challenge #15 from [2] and its solution.

**Constraints:** The $\text{START}(D)$ function creates a variable $X_i$ for each vertex pair in $G$ in horizontal or vertical line-of-sight, with domain $\{0,1,2\}$ indicating the number of edges between them. A $\text{SUM_EQUALS}(h)$ constraint is created for each vertex to define the target number of coincident edges. A $\text{PRODUCT_EQUALS}(0)$ constraint is created for each potential edge crossing to ensure that no edge crosses any other. A $\text{CONNECTED}$(vertices) constraint is created to ensure that all vertices remain (potentially) connected.

**Results:** The results are shown in Table III. The Nikoli challenges were taken from the Nikoli books Hashiwokakero 1 [20] and Hashiwokakero 2 [21] and the Times examples from the The Times book of Japanese Logic Puzzles [2].

Again, the $diff$ ratings produced by DS are reasonably consistent with each publisher’s ranking in order of difficulty. All examples are solved with a 1-LoE search, which suggests that Hashiwokakero is somewhat “flat” in nature, with decisions being more immediate rather than requiring significant (embedded) deduction. The true complexity of this puzzle for players may lie in the difficulty of mentally untangling connected sets within convoluted graphs, a task more suited to computation than the human brain. Interestingly, the Hashiwokakero 1 examples appear harder than the corresponding Hashiwokakero 2 examples, as the former all contain 0-LoE instantiations while the latter do not.

D. Zebra Puzzle

The Zebra Puzzle, also known as Einstein’s Puzzle, is a traditional logic puzzle in which the player must deduce the answers to certain questions based on given statements [22]. It is recognised as a difficult example of its type, which only an estimated 2% of the population can solve.

The statements are:

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in a house next to the man with the fox.
12. Kools are smoked in a house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
15. The Norwegian lives next to the blue house.

The questions are:

1. Who drinks water?
2. Who owns the zebra?

This is an older form of logic puzzle, and is probably what most players would have understood a “logic puzzle” to be before the recent advent of Japanese logic puzzles. We apply DS to test its operation on this more traditional example.

**Constraints:** The $\text{START}(D)$ function creates a $5 \times 5$ table of variables $X_i$ with domain $\{\ldots, 5\}$, in which the rows represent the categories and the columns represent each house from left to right. An $\text{ALL_DIFFERENT}$ constraint is created for each row, and a $\text{RELATION}$ constraint is created for each
statement. Statements \#9 and \#10 are immediately instantiated as known facts (hints).

**Results:** The results are shown in Table IV. The first question is answered with 1-LoE DS (the Norwegian drinks water) while the second answer requires 2-LoE DS (the Japanese owns the zebra). The relatively high \textit{diff} score may go some way to explaining why this puzzle is deemed so hard for humans.

| Challenge | Size | \(|X|\) | \(|Y|\) | \(|H|\) | \(s\) | \(S_0\) | \(S_1\) | \(V_1\) | \(A_1\) | \(S_2\) | \(V_2\) | \(A_2\) | \textit{diff} |
|-----------|------|--------|--------|--------|----|------|------|------|------|------|------|------|--------|
| Zebra Puzzle | 5 \times 5 | 25 | 17 | 2 | 0.011 | 1 | 4 | 0 | 6 | 3 | 9 | 3.120 |

**Table IV. Zebra Puzzle Results.**

**E. Pentominoes**

Pentomino packings are a geometric puzzle in which players must pack the twelve \textit{pentominoes} (Figure 11) into a shape [23]. The task modelled here is to find all deducible two-piece challenges for the 6 \times 10 packing shown in Figure 13.

**Constraints:** The \textit{Start(D)} function creates a variable \(X_i\) for each of the 12 tiles, with domains ranging in size from 32 to 304 depending on the number of possible placements of each piece. A \textit{NoCollision} constraint is created and initialised with bits corresponding to the cells occupied by each potential placement, to ensure that no pieces intersect. A \textit{Covers(cells)} constraint is created to ensure that every cell is (potentially) occupied by at least one piece.

**Results:** Table VI shows the results for all deducible two-piece challenges for the specified 6 \times 10 packing.

| Challenge | Size | \(|X|\) | \(|Y|\) | \(|H|\) | \(s\) | \(S_0\) | \(S_1\) | \(V_1\) | \(A_1\) | \(S_2\) | \(V_2\) | \(A_2\) | \textit{diff} |
|-----------|------|--------|--------|--------|----|------|------|------|------|------|------|------|--------|
| UY | 6 \times 10 | 12 | 2 | 2 | 1.379 | 0 | 0 | 0 | 6 | 1 | 3 | 1.757 |
| PQ | 6 \times 10 | 12 | 2 | 2 | 1.644 | 0 | 0 | 0 | 6 | 1 | 3 | 2.295 |
| PR | 6 \times 10 | 12 | 2 | 2 | 0.598 | 1 | 8 | 1 | 0 | 0 | 0 | 0.190 |
| PT | 6 \times 10 | 12 | 2 | 2 | 0.221 | 0 | 7 | 3 | 0 | 0 | 0 | 1.467 |
| PU | 6 \times 10 | 12 | 2 | 2 | 0.226 | 0 | 9 | 1 | 0 | 0 | 0 | 1.589 |
| PY | 6 \times 10 | 12 | 2 | 2 | 0.260 | 0 | 0 | 0 | 7 | 1 | 2 | 1.118 |
| PZ | 6 \times 10 | 12 | 2 | 2 | 0.145 | 0 | 8 | 2 | 0 | 0 | 0 | 0.247 |
| RY | 6 \times 10 | 12 | 2 | 2 | 0.051 | 1 | 7 | 2 | 0 | 0 | 0 | 0.689 |
| SY | 6 \times 10 | 12 | 2 | 2 | 0.060 | 1 | 7 | 2 | 0 | 0 | 0 | 0.971 |
| UY | 6 \times 10 | 12 | 2 | 2 | 0.138 | 0 | 9 | 1 | 0 | 0 | 0 | 1.062 |
| YZ | 6 \times 10 | 12 | 2 | 2 | 0.396 | 0 | 0 | 0 | 5 | 1 | 4 | 0.626 |

**Table VI. Pentominoes Results.**

Figure 12 shows easy (UY) and harder (PQ) challenges, according to DS. PQ does not allow any easy 0-LoE or 1-LoE deductions so the solver has no real starting point, whereas UY offers easier deductive purchase. Figure 13 shows the key deduction \(S\) and the deduction order of the remaining placements. Note that the \textit{diff} scores do not necessarily correlate with the \(S_2, V_2\) and \(A_2\) counts shown, as \textit{diff} also includes the (considerable) number of eliminations involved.

**IV. Strategic vs Deductive Depth**

Consider the Slitherlink example shown in Figure 14, in which a 1-LoE DS has been completed and a 2-LoE DS is required to make further progress. From the Jordan Curve Theorem, each cell must be either inside or outside the closed solution path [18]. Those cells known to be inside the path are coloured dark grey, those known to be outside are white, and those not yet known are light grey.

A further \textit{Connected(cells)} constraint can be added which will make the key deduction (indicated) and allow 1-LoE solution, as the alternative would disconnect the lower right coloured region. Hence a \textit{colouring} strategy used by players can be incorporated into the search as a constraint to facilitate easier solution. This constraint saves a level of embedding in this case, but is expensive to compute and is only occasionally useful for some near-complete solutions.

The actual payoff of each constraint must be weighed against its cost, although on balance this constraint would probably be added if the purpose of the solver was to find instances that require particular player strategies. DS could be tried with each constraint turned off one by one, to identify those challenges that require a broader range of solution strategies, and are hence more likely to be of interest to players.

This example also demonstrates a new way of looking at puzzles through DS. Figure 15 shows the \textit{deduction profile} given by the number of variables instantiated per iteration of DS for the Janko #100 challenge. The cycles of peaks and troughs reveal a repeated process of many instantiations reducing to crisis points, in which the solver must make one or two key deductions to “open up” the puzzle again and allow progress to the next cycle of deductions. This is reminiscent of the peaks of \textit{tension} and subsequent troughs of relaxation found in well-designed board games [24], indicating that this challenge may have a good “shape” for players.
LoE Sudoku cases) suggests that deduction puzzles written by humans, for humans, generally involve no more than two levels of recursive embedding. This lends weight to our initial conjecture that deduction puzzles should generally involve no more than two levels of recursive embedding if they are to be human-solvable.

Future work might include complexity analysis of the algorithm and deeper comparisons with related SAT and CSP algorithms such as AC-3. User studies are needed to gauge the accuracy of \textit{diff} estimates in the eyes of actual players.

ACKNOWLEDGMENT

Thanks to Stephen Tavener for discussions. This work was partly funded by EPSRC grant EP/I001964/1.

REFERENCES