Solving *Sudoku* With Ant Colony Optimization

Huw Lloyd, *Member, IEEE*, and Martyn Amos

Abstract—**In this article, we present a new algorithm for the well-known and computationally challenging *Sudoku* puzzle game. Our ant-colony-optimization-based method significantly outperforms the state-of-the-art algorithm on the hardest, large instances of *Sudoku*. We provide evidence that—compared to traditional backtracking methods—our algorithm offers a much more efficient search of the solution space, and demonstrate the utility of a novel antistagnation operator. This work lays the foundation for future work on a general-purpose puzzle solver, and establishes Japanese pencil puzzles as a suitable platform for benchmarking a wide range of algorithms.**

Index Terms—**Ant colony optimization, puzzle games, *Sudoku*.**

	6					5		2
	3				7			
	2		3		6			7
8	7	3		2	1	4	5	
9	4		5					
	1				4			
				9	5			4
3	9	4	8	1			7	5
	5	1		6	3		9	8

4	6	7	1	8	9	5	3	2
1	3	8	2	5	7	9	4	6
5	2	9	3	4	6	1	8	7
8	7	3	6	2	1	4	5	9
9	4	2	5	3	8	7	6	1
6	1	5	9	7	4	8	2	3
2	8	6	7	9	5	3	1	4
3	9	4	8	1	2	6	7	5
7	5	1	4	6	3	2	9	8

Fig. 1. Structure of a *Sudoku* puzzle instance (left), and its solution (right).

I. Introduction

SUDOKU is a well-known logic-based puzzle game that was first published in 1979 under the name of *Number Place*. It was popularized in Japan in 1984 by the puzzle company Nikoli, and later named *Sudoku*, which roughly translates to "single digits." The puzzle gained attention in the West in 2004, after *The Times* published its first *Sudoku* grid at the instigation of Hong Kong-based judge Wayne Gould, who first encountered the puzzle in 1997, and developed a computer program to automatically generate instances. *Sudoku* is now a global phenomenon, and many newspapers now carry it alongside their existing crosswords (see [1] for a general history of the puzzle).

The simplest variant of *Sudoku* uses a 9×9 grid of cells divided into nine 3×3 subgrids [Fig. 1 (left)]. As we later demonstrate in Section III, the problem scales to larger grids, but, for the moment, we focus on the most familiar variant. The aim of the puzzle is to fill the grid with digits such that each row, each column, and each 3×3 subgrid contains all of the digits 1–9 [Fig. 1 (right)]. An instance of *Sudoku* provides, at the outset, a partially completed grid, but the difficulty of any grid derives more from the range of techniques required to solve it than the number of cell values that are provided for the player.

Formally, a *Sudoku* problem of order $n = 3$ is made up of a grid of *cells* (or squares), arranged into 3×3 subgrids known as *boxes*. A *unit* is a row, column, or box, each containing exactly nine cells. A problem is solved when *each unit* (that is, every row, column, and box) contains a permutation of the digits $1 \ldots 9$ [2].

Any given cell has exactly three units and 20 *peers*; the units are the row, column, and box in which the cell resides, and the set of peers is made up of the other cells in those units (that is, $2 \times 8 = 16$ neighbors in the relevant row and column, plus 4 other cells occupying the same box; see Fig. 2).

Sudoku is an NP-complete problem [3], as first shown in [4] via a reduction from the Latin Square Completion problem [5]. As such, the problem offers itself as a useful benchmark challenge, and a number of different types of algorithm have been proposed for its solution (see Section II for a more detailed discussion of these). However, we also consider the argument that "We should develop AI methods that work with not just one game, but with any game (within a given range) that the method is applied to" [6]. That is, rather than developing a multitude of algorithms to play one specific game, we should seek methods that find broader applicability, across a *range* of games. Although the algorithm we present here is demonstrated in the context of *Sudoku*, we later show in Section V, how its lack of reliance on any heuristic information (that is, game-specific "hints") means that it may be applied to a number of different puzzle games.

While such puzzle games may, superficially, appear to lack "real world" relevance, they in fact offer a significant challenge for general-purpose AI methods; as argued in [6], "We need game playing benchmarks and competitions capable of expressing any kind of game, including puzzle games, two-dimensional (2-D) arcade games, text adventures, 3-D action-adventures, and so on; this is the best way to test general AI capacities and reasoning skills." While our algorithm could not be described as "general purpose," this does serve to underscore the importance of the puzzle game domain.

The rest of this article is structured as follows. In Section II, we briefly review closely related recent work on the application of

Manuscript received August 13, 2018; revised February 19, 2019 and June 18, 2019; accepted September 13, 2019. Date of publication September 20, 2019; date of current version September 15, 2020. *(Corresponding author: Huw Lloyd.)*

H. Lloyd is with the Department of Computing and Mathematics, Manchester Metropolitan University, Manchester M15 6BH, U.K. (e-mail: huw.lloyd@mmu.ac.uk).

M. Amos is with the Department of Computer and Information Sciences, Northumbria University, Newcastle upon Tyne NE1 8ST, U.K. (e-mail: martyn.amos@northumbria.ac.uk).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TG.2019.2942773

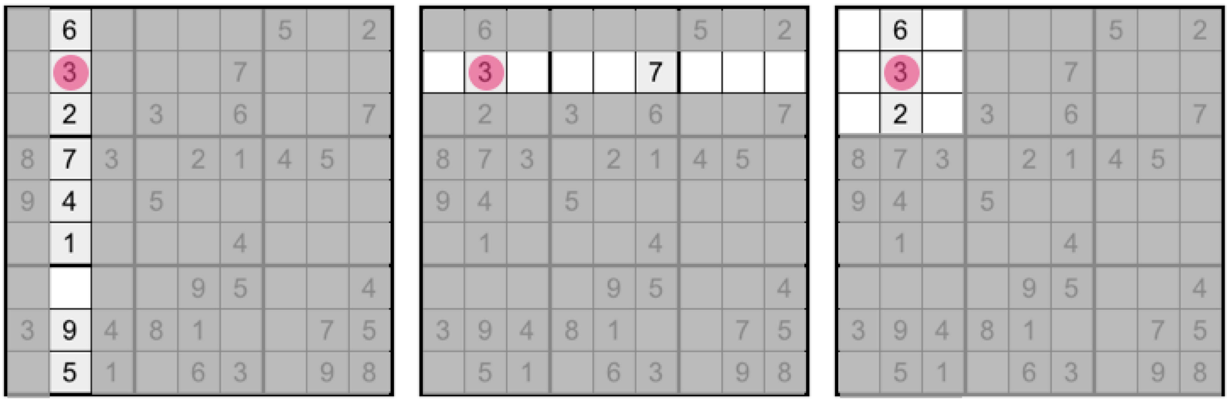


Fig. 2. Units and peers for a specific highlighted cell. The *units* (from right to left, column, row, and block) are highlighted in white. The union of the three units, that is all the white cells, are the *peers*.

various algorithms to *Sudoku*. This motivates the description, in Section III of our own method, based on ant colony optimization (ACO), which introduces a novel operator which we call best value evaporation (BVE). In Section IV, we present the results of experimental investigations, which confirm: 1) that our algorithm outperforms existing methods and 2) that BVE is a necessary addition to the basic ACO algorithm for solving large *Sudoku* instances. We conclude in Section V with a discussion of our findings, and discuss possible future work in this area.

II. Related Work

We first consider a "traditional" backtracking approach to solving *Sudoku*. The exact cover problem [7] is a type of constraint satisfaction problem which may be phrased as follows: given a binary matrix, find a *subset* of rows in which each column sums to 1 (that is, find a set of rows in which each column contains only a single 1). In [8], Knuth describes the "dancing links" implementation of his Algorithm X (called DLX), a "brute force" backtracking algorithm for exact cover. As any *Sudoku* puzzle may be transformed into an instance of Exact Cover [9], DLX naturally offers an effective solution method for *Sudoku* [10].

In [2], Norvig presents an alternative approach, based on constraint propagation (CP) followed by a search process (we discuss this in more detail shortly). Other notable approaches to solving *Sudoku* include formal logic [11], an artificial bee colony algorithm [12], constraint programming [13], [14], evolutionary algorithms [15]–[18], particle swarm optimization [19], [20], simulated annealing [21], tabu search [22], and entropy minimization [23]. As this diverse set of solution methods demonstrates, *Sudoku* offers a challenging yet conceptually simple test-bed for the comparative analysis of algorithms for problems involving complex reasoning.

In this article, we focus on the application of ACO to the solution of *Sudoku*. ACO is a population-based search method inspired by the foraging behavior of ants [24], [25], and it has been successfully applied to a wide range of computational problems (see [26] and [27] for overviews of both the algorithm and its applications). The basic ACO algorithm uses a population of "ants" (agents), which individually explore a given problem space and incrementally construct a solution, combined with a global "pheromone" data structure, which is used to inform decisions taken by the ants. Essentially, each ant moves individually on some problem representation (for example, a graph), gradually building a solution and probabilistically choosing its next move according to pheromone concentrations. Components with more pheromone are more likely to be selected. After a single population iteration, the best solution according to some objective function is selected from the population, and the components it contains (e.g., edges in the graph) are given additional pheromone. In this way, the population rapidly converges on high-quality solutions, although premature or suboptimal convergence is discouraged through the continuous "evaporation" of pheromone concentrations. Some ACO variants include *local* pheromone operators, which allow individual ants to record information about their traversal during the solution construction process; for example, ants may reduce the global pheromone value associated with components as they are added to a solution, to discourage following ants from taking the same path.

The archetypal ACO algorithm was named "ant system" [25], and this was applied to the well-known traveling salesman problem as follows: each edge connecting two cities has a pheromone value, and the probability of an edge being selected by an ant is a function of both its pheromone concentration and its distance from the ant's current location. This process thus combines the autocatalytic power of the global pheromone network with a greedy local search heuristic. Each ant also maintains a "tabu" list of cities that it has visited, and an ant may not revisit any city on its list. Once it has visited all cities, an ant then deposits an amount of global pheromone which is inversely proportional to the length of its tour; that is, shorter tours deposit more pheromone. Once all ants have completed this process, the global pheromone matrix is evaporated, thus gradually removing the remnants of suboptimal tours that persist over time. Dorigo *et al.* [25] demonstrate that positive feedback, combined with local search, can offer a heuristic that is robust, versatile, broadly applicable, and amenable to parallelization, because of its inherent population-based structure. Since the publication of the original paper, ACO is now a well-established method [28].

In [29], Mantere presents a hybrid ACO/genetic algorithm approach to *Sudoku*, which combines global (evolutionary) search with greedy local (ACO-based) search. Schiff [30] and Sabuncu [31] also present relatively recent work on applying ACO to *Sudoku*, but, in both cases, the performance of the algorithm is relatively poor. Another nature-inspired approach was used by [12], who used a variant of the artificial bee colony algorithm to solve 9×9 *Sudoku* puzzles. The algorithm was able to solve some difficult instances (such as the *AIEscargot* instance [32]) but the runtime performance is relatively poor with an average solution time of over 6 min for difficult instances.

For the purposes of comparison, in this paper we focus mainly on the work of Musliu *et al.* [33], who present an iterated local search (ILS) algorithm with constraint programming which represents the state of the art in stochastic search algorithms for the *Sudoku* problem, plus the algorithms of Knuth [8] and Norvig [2].

III. Our Algorithm

In [2], Norvig describes a two-component approach to solving *Sudoku*, using a combination of CP and *search*. CP ensures that the "rules" of *Sudoku* are observed, and repeatedly prunes the *value set* of each cell (that is, the set of possible values that cells might take). Importantly, by using CP during search, we effectively "parallelize" the process, by eliminating large numbers of possible cell values every time we fix a cell's value; selecting a specific value for a cell immediately rules out that value's presence in a large number of other cells. In [2], Norvig combines CP with a recursive depth-first search which, at each iteration, selects the cell with the smallest value set, and then chooses the first numeric ordered value for that cell. This essentially maximizes the probability of "guessing correctly," and is referred to as the minimum remaining values heuristic.

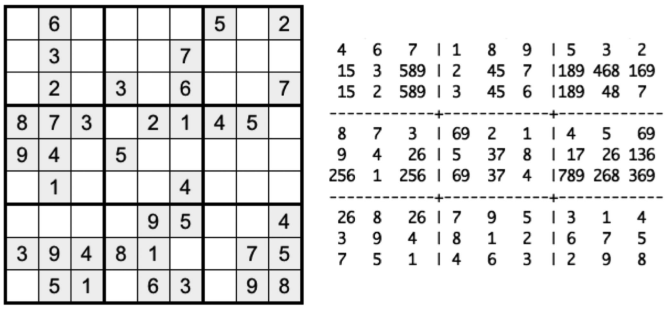
Here, we present a variant of CP inspired by Norvig's method, and use ACO (rather than depth-first search) to search the space of solutions. We now describe our CP method in more detail. For clarity, this is written in terms of the 9×9 *Sudoku* puzzle, but the method generalizes trivially to larger sizes (e.g., 16×16, 25×25).

A. Constraint Propagation

Throughout the CP process, each cell maintains its value set—a list of *possible values* it might take; every cell starts with the same value set, $[1 \ldots 9]$. Once a set has been reduced to a single value, we call that value *fixed* for that cell. Our CP algorithm implements two basic rules, which are applied to a cell's peers when it has its value fixed:

1) eliminate from a cell's value set all values that are *fixed* in any of the cell's *peers*;
2) if any values in a cell's value set are in the only possible place *in any of the cell's units*, then fix that value.

Note that since this can lead to other cells having their values fixed, the procedure is recursive, and terminates when no further changes are possible.



	6					5		2
	3				7			
	2		3		6			7
8	7	3		2	1	4	5	
9	4		5					
	1				4			
				9	5			4
3	9	4	8	1			7	5
	5	1		6	3		9	8

```
 4   6   7  | 1   8   9  | 5   3   2
 15  3  589 | 2   45  7  |189 468 169
 15  2  589 | 3   45  6  |189  48  7
------------+------------+------------
 8   7   3  | 69  2   1  | 4   5   69
 9   4   26 | 5   37  8  | 17  26 136
256  1  256 | 69  37  4  |789 268 369
------------+------------+------------
 26  8   26 | 7   9   5  | 3   1   4
 3   9   4  | 8   1   2  | 6   7   5
 7   5   1  | 4   6   3  | 2   9   8
```

Fig. 3. Instance from Fig. 1 (left), and (right) cell value sets after initial pass of CP algorithm. The value sets are represented as strings of allowable digits for the cell, for example "589" represents the set of values $\{5, 8, 9\}$.

In Fig. 3, we show the instance from Fig. 1 after the *initial* pass of our CP algorithm, which occurs when the board is set up, and before any search is performed. For easy cases, the application of the CP algorithm is often sufficient to solve the board, and no further search is required (see Section IV for a discussion). However, in most cases, some search will be required, and we now describe our ACO-based method for this.

B. Our ACO Algorithm

Our algorithm is based on ant colony system (ACS), which is a variant of ACO introduced in [34]. We first give an informal description of the algorithm, and then formally specify its various components.

At each population-level iteration, every ant works independently on its own copy of the board. However, the global pheromone matrix persists across iterations, allowing for a combination of local search and global positive feedback to occur (i.e., when the best ant in each iteration updates the global pheromone). The ants move round their boards *in parallel*; the ant system iterates over the ants in turn, calling a step function which moves each ant one step. This enables ants to discourage others from following the same path through the local pheromone mechanism. The outer loop of the ant system update therefore iterates c times, where c is the number of cells, and at each iteration requests that each ant makes a single step.

As previously stated, once the initial pass of the CP has been completed, then most cells will have a set of possible values. The aim of each ant, in a single population-level iteration, is to fix as many cell values as possible. Each ant starts on a different, randomly selected cell, and then iterates over all cells on the board. We simply move from one cell to the next because what is important is not the "next cell," but the *value assigned* to the next cell encountered. Whenever it leaves a cell that does not have a fixed value (that is, a cell with a number of possible values), an ant must make a decision on which element of that cell's value set to choose, thus setting the cell to that value. Importantly, as soon as an ant sets the value of a cell, the constraints that it introduces are propagated across the board.

Decisions on which value to choose are based on relative pheromone levels, which are assigned to each possible value. These are stored in a *pheromone matrix*, which keeps track

Algorithm 1: Our ACO Algorithm for Sudoku.

```
1  read in puzzle;
2  for all cells with fixed values do
3      propagate constraints (according to Section III-A);
4  end
5  initialize global pheromone matrix;
6  while puzzle is not solved do
7      give each ant a local copy of puzzle;
8      assign each ant to a different cell;
9      for number of cells do
10         for each ant do
11             if current cell value not fixed then
12                 choose value from current cell's value set;
13                 fix cell value;
14                 propagate constraints;
15                 update local pheromone;
16             end
17             move to next cell;
18         end
19     end
20     find best ant;
21     do global pheromone update;
22     do best value evaporation;
23 end
```

of a single pheromone amount for each possible value in each cell. This is, for an order-3 (9×9) *Sudoku* puzzle, a matrix of 81×9 values, with each cell corresponding to the pheromone level for each possible value ($1 \ldots 9$) in a cell (indexed $1 \ldots 81$). Depending on the "greediness" of the selection, either the value with the highest pheromone value is chosen, or a weighted (roulette) selection is made.

After the cell's value is set, the standard ACS *local* pheromone operator is applied, which reduces the probability of that value being selected by the following ant, thus preventing early convergence.

Once all ants have covered every square of the board, we then perform the *global* pheromone update, which rewards only the best solution found so far (the *global best*, in line with ACS principles). We characterize the "best" solution, at each iteration, as the sequence of value selections that lead to the greatest number of cells having their values fixed; the best solution is effectively the one found by the ant that "guesses" correctly the highest number of times. However, at this point, we introduce a novel variation to the standard ACS algorithm, which we call BVE. In what follows, "best value" refers to an amount of pheromone that is added to the global pheromone matrix whenever the best solution is identified within a generation, and this value is itself subject to evaporation, along with the component pheromone values.

In *standard* ACS, the global pheromone operator increases the pheromone concentrations of all components of the global best solution with an amount of pheromone that is directly proportional to the *absolute quality* of that solution. However, this can gradually lead to *stagnation*, where all ants end up selecting the same route. Instead, the amount of pheromone that is added globally, which we call the *best value*, is measured in terms of the *proportionate quality* of the best solution found so far (5). Importantly, the best value itself is subject to evaporation over time, which prevents "lock in"; taken together, these two components of BVE prevent premature stagnation, which is confirmed by our later experimental observations.

We give a pseudocode description of our approach in Algorithm 1, components of which we now formally specify.

Line 5: For a *Sudoku* puzzle of dimension d, we define a 2-D global pheromone matrix τ in which each element is denoted as τ_i^k, where i is the cell index ($1 \le i \le d^2$) and k is a possible value for the cell ($k \in [1, d]$). τ_i^k represents the pheromone level associated with value k in cell i. Each element of the matrix is initialized to some fixed value τ_0 (we use a value of $1/c$, where $c = d^2$ is the total number of cells on the board).

Line 12: Where an ant has a choice of a number of values in an "open" cell (i.e., one which does not yet have its value fixed), then we define the *value set* v_i of cell i as the set of all available values for that cell, from which we have to choose one. We have a choice of two methods to use when making a selection; we might make a *greedy* selection, in which case the member of v_i with the highest pheromone concentration is selected, or we might make a *weighted* (i.e., "roulette wheel") selection, in which case the selection probabilities are proportional to the pheromone associated with the available choices. The relative probabilities of each type of selection are determined by the *greediness* parameter, $q_0 \in [0, 1]$. A value selection s is, therefore, made according to

$$s = \begin{cases} \operatorname{argmax}_{k \in v_i} \{\tau_i^k\}, & \text{if } q < q_0 \\ R, & \text{otherwise} \end{cases} \quad (1)$$

where $q \in [0, 1]$ is a uniform random deviate, and R is a selection from v_i made according to the probability distribution

$$p_i^k = \frac{\tau_i^k}{\sum_{j \in v_i} \tau_i^j}, \quad k \in v_i \quad (2)$$

where p_i^k is the probability of selecting choice k from v_i.

If a cell has a value set of size zero (that is, it cannot have its value fixed due to other cells being fixed and the constraints thus introduced), then we mark it as a "fail cell"; the number of fail cells is later subtracted from the number of cells to be fixed when we calculate the quality of a solution (see the note for line 20).

Line 15: The local pheromone update operator is used to make selected values less attractive in subsequent iterations, thus promoting exploration of the solution space. The local pheromone update is handled as follows; every time an ant selects a value s at cell i, its pheromone value in the matrix is updated as follows:

$$\tau_i^s \leftarrow (1 - \xi)\tau_i^s + \xi\tau_0 \quad (3)$$

with $\xi = 0.1$ (the standard setting for ACS).

Line 20: In order to perform the global pheromone update, we must first find the best performing ant. At each iteration, each ant n of the m ants keeps track of the number of cells,

$f_n, n \in \{1 \ldots m\}$, that it has managed to set to a specific value. The value of f_n corresponding to the iteration-best ant is f_{best}, given by

$$f_{\text{best}} = \max_{n \in \{1 \ldots m\}} f_n. \tag{4}$$

We then calculate the amount of pheromone to add $\Delta\tau$ as follows:

$$\Delta\tau = \frac{c}{c - f_{\text{best}}} \tag{5}$$

where c is the total number of cells on the board. If the value of $\Delta\tau$ exceeds the current "best pheromone to add" value $\Delta\tau_{\text{best}}$ (a quantity initialized to 0 at the beginning of the run), then we set $\Delta\tau_{\text{best}} \leftarrow \Delta\tau$, and replace the current best solution with the solution found by the iteration-best ant.

Line 20: We then update all pheromone values corresponding to values in the current best solution, where $\rho \in [0, 1]$ is the standard evaporation parameter

$$\tau_i^s \leftarrow (1 - \rho)\tau_i^s + \rho\Delta\tau_{\text{best}}. \tag{6}$$

Note that in ACS, there is no global evaporation of pheromone; the global pheromone update (6) is only applied to pheromone values corresponding to fixed values in the *best* solution; the evaporation parameter ρ represents the "volatility" of the deposited pheromone, and is used to tune the convergence rate of the algorithm.

Line 22: In order to prevent "lock in," we then additionally apply evaporation to the current best pheromone value $\Delta\tau_{\text{best}}$

$$\Delta\tau_{\text{best}} \leftarrow \Delta\tau_{\text{best}} \times (1 - \rho_{\text{BVE}}) \tag{7}$$

where $\rho_{\text{BVE}} \in [0, 1]$ is a parameter which controls the rate of evaporation of the best pheromone value.

IV. Experimental Results

Our ant colony algorithm (ACS) was evaluated by comparing it with 1) ILS code from Musliu *et al.* (ILS) [33]; 2) a C++ implementation of the dancing links algorithm (DLX) [35]; and 3) our own implementation of backtracking search (BS), using the minimum remaining values heuristic, which uses the same problem representation and CP code as the ant colony algorithm (BS). The code presented in [33] was itself compared against a number of other stochastic algorithms, and was shown to be the best performing. We include the dancing links and backtracking algorithms for comparison with deterministic, exhaustive search. Furthermore, including a BS which uses the same underlying CP code allows us to evaluate the effectiveness of the ant colony algorithm in searching the problem space, independent of the details of the underlying implementation.

We conducted experiments using a number of logic-solvable 9×9 instances from the literature, as well as randomly generated 9×9, 16×16, and 25×25 "general" instances (which do not necessarily have a unique solution). In all the experiments, we evaluated the algorithms for success rate over a number of trials or instances, subject to a timeout, and the mean time to solution. This is the same as the evaluation conducted by [33] of their algorithm against a number of competitors, and gives a measure of the practical applicability of the algorithm in a time-constrained environment. In all cases, we measured the statistical significance of results using nonparametric tests, with a p value threshold for significance of 0.05. In cases where multiple algorithms are compared together, this significance threshold was modified using the Bonferroni correction. In comparing vectors of solution times, we use the Mann–Whitney U test in cases where the vectors have different lengths, which occurs when the success rates in an experiment differ. This test is appropriate for determining significance of differences in the means of differently sized samples, when the distribution cannot be assumed to be normal. In cases where all algorithms solved all the instances, we use the Wilcoxon-signed rank test, which tests for significance of difference in the means of paired observations, again with no assumption on the distribution. The success rates are treated as frequencies of a nominal variable (success/fail) for which the Pearson χ^2 test is appropriate.

A. Experimental Environment

All of the codes were compiled using the same compiler and optimization setting (g++ v5.4.0 with -O3). Experiments were run on a machine with an Intel Xeon E5-2460v4 processor with a clock speed of 2.4 GHz, running Ubuntu Linux. The parameter settings for the ILS solver were taken from the recommendations given in [33]. For the ant colony code (ACS), we used the following settings: $\rho = 0.9$, $q_0 = 0.9$, $\rho_{\text{BVE}} = 0.005$, $m = 10$. Our code, and all the instance files used for the experiments, may be downloaded from https://github.com/huwlloyd-mmu/sudoku_acs.

B. Logic-Solvable 9×9 Instances

We first selected instances based on known difficulty, or on previous use in the literature. We selected the ten instances used in [31] (labeled here *sabuncu1–sabuncu10*), five named instances identified by [36] as the most difficult (*Platinum Blond, Golden Nugget, Red Dwarf, coly013, tarx0134*), and one instance (*AI Escargot*) [32], commonly regarded as an extremely difficult puzzle. These instances are all *logic solvable*; in other words, they each have a unique solution which can be deduced from the given numbers. We ran the ACS, ILS, dancing links (DLX), and BS algorithms 100 times on each instance, with a timeout of 5 s. The puzzles were successfully solved in all cases by all four algorithms; there were no timeouts. Table I shows the timing results for the four algorithms. Since all the instances were solved in all cases, the vectors of times per instance and algorithm are the same length; we therefore use the Wilcoxon-signed rank test, to determine the significance of differences in the mean time. In all cases, we tested the fastest algorithm against the other three, using the Bonferroni correction to lower the significance threshold on the p value of the tests by a factor equal to the number of tests. We also tested the times obtained by the two stochastic algorithms, ACS and ILS, against each other.

The ten puzzles from [31] (*sabuncu1–sabuncu10*) are generally solved in less time by all the algorithms than the six

TABLE I
SOLUTION TIMES (MEAN AND STANDARD DEVIATION TIME OVER 100 RUNS) FOR THE LOGIC-SOLVABLE INSTANCES

	Solution Time/s			
Instance	ACS	ILS	DLX	BS
sabuncu1	$(4.8 \pm 1.84) \times 10^{-5}$*	0.00083 ± 0.00047	0.00105 ± 0.000362	$\mathbf{(1.58 \pm 0.651) \times 10^{-6}}$
sabuncu2	$(4.82 \pm 1.73) \times 10^{-5}$*	0.00414 ± 0.00143	0.000937 ± 0.000308	$\mathbf{(2.18 \pm 6.45) \times 10^{-6}}$
sabuncu3	0.000993 ± 0.000457*	0.112 ± 0.0296	0.00104 ± 0.000366	$\mathbf{0.000202 \pm 0.0000775}$
sabuncu4	0.000625 ± 0.000708*	0.00859 ± 0.00229	0.00112 ± 0.000346	$\mathbf{0.0001 \pm 0.0000378}$
sabuncu5	$(4.62 \pm 1.5) \times 10^{-5}$*	0.00097 ± 0.000556	0.00101 ± 0.000384	$\mathbf{(1.68 \pm 0.733) \times 10^{-6}}$
sabuncu6	0.0107 ± 0.00828*	0.105 ± 0.027	0.00153 ± 0.00045	$\mathbf{0.000775 \pm 0.000273}$
sabuncu7	0.00106 ± 0.000986*	0.0853 ± 0.0206	0.00102 ± 0.000318	$\mathbf{(9.67 \pm 3.75) \times 10^{-5}}$
sabuncu8	0.000728 ± 0.000343*	0.007 ± 0.00206	0.00107 ± 0.000374	$\mathbf{(7.91 \pm 2.74) \times 10^{-5}}$
sabuncu9	0.00163 ± 0.0014*	0.0153 ± 0.00437	0.00105 ± 0.000345	$\mathbf{0.00016 \pm 0.0000579}$
sabuncu10	$(4.73 \pm 1.85) \times 10^{-5}$*	0.00136 ± 0.000641	0.00104 ± 0.000363	$\mathbf{(1.6 \pm 0.693) \times 10^{-6}}$
aiescargot	0.0204 ± 0.0152*	0.152 ± 0.0328	0.00208 ± 0.000648	$\mathbf{0.000475 \pm 0.000182}$
coly013	0.0488 ± 0.0518*	0.702 ± 0.0685	$\mathbf{0.007 \pm 0.00146}$	0.0278 ± 0.00517
goldennugget	0.0374 ± 0.0293*	0.442 ± 0.0918	$\mathbf{0.00545 \pm 0.00149}$	0.0152 ± 0.00304
platinumblond	0.113 ± 0.0859*	0.131 ± 0.0223	0.0059 ± 0.00152	$\mathbf{0.00268 \pm 0.000923}$
reddwarf	0.0404 ± 0.0354*	0.299 ± 0.0768	$\mathbf{0.00514 \pm 0.00132}$	0.00993 ± 0.00212
tarx0134	0.0259 ± 0.0193*	0.851 ± 0.0699	$\mathbf{0.0185 \pm 0.00303}$	0.038 ± 0.0074

Figures in bold indicate times which are significantly lower for one algorithm compared the other three, based on a Wilcoxon-signed rank test; the Bonferonni correction is applied, so that p values less than $0.05/3$ are taken to be significant. Asterisks show cases in which the times for ILS or ACS are significantly lower than the other, using the Wilcoxon-signed rank test with $p < 0.05$.

harder puzzles. In four cases (*sabuncu1, sabuncu2, sabuncu5*, and *sabuncu10*) the puzzle is solved by a single application of our CP procedure, so that no searching is required for either the ACS or BS algorithms. The difference in runtimes between the two algorithms for these instances may be explained by the difference in setup times; in the case of ACS, the overhead of creating the ant colony and initializing the pheromone matrix is clearly significant. On these four "trivial" instances, the BS algorithm is the fastest of all (running in times of order a microsecond). DLX requires at least of order a millisecond to solve all the puzzles; in all but the most difficult cases, this time is most likely dominated by the calculations to convert the instance to and from an instance of the exact cover problem.

Overall, we find that the deterministic solvers perform best on these instances. Either DLX or BS is significantly fastest for all of the instances. BS is the best performing overall, and is fastest in 12 of the 16 instances, with DLX fastest in the other four. ACS is significantly faster than ILS in all cases, and faster than DLX in seven of the 16 instances.

Finally, we note that the times reported by Sabuncu [31] for their ACO algorithm to solve ten of the instances used here are typically 1–3 s. This is several orders of magnitude slower than our times using ACS for the same instances which are of the order of milliseconds, or less; this is more than can be accounted for by differences in hardware or efficiency of implementation and although we have not performed a direct comparison with their code, we can safely assume that our algorithm is the better performing of the two.

C. General Instances

Following [14] and [33], we generated random instances for the 9×9, 16×16, and 25×25 *Sudoku* problem. In the latter two cases, subgrids are of size 4×4 and 5×5, respectively, and each row, column, and subgrid must contain all of the digits $1 \ldots 16$ and $1 \ldots 25$, respectively.

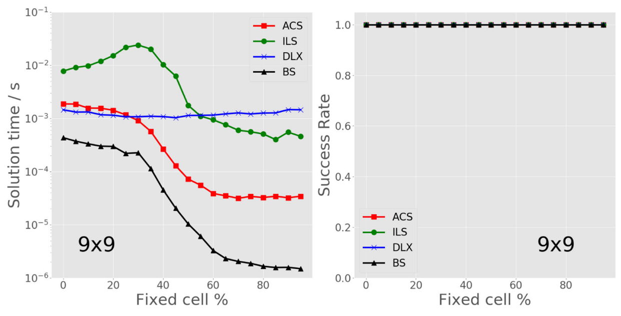


Fig. 4. Plots of solution time (left) and success rate (right) against fixed cell percentage for runs of ACS, ILS, DLX, and BS on the 9×9 general instances.

These instances were generated by running the ACS code with an initially blank grid, to produce a set of *Sudoku* solutions. These are then converted into problem instances by randomly blanking a number of the cells. The instances generated in this way are not guaranteed to have a unique solution. For each of the sizes 9×9, 16×16, and 25×25, we generated 100 instances for fixed cell fractions in steps of 0.05 from 0 to 0.95, giving a total of 6000 individual instances. We ran the ACS, ILS, DLX, and BS codes once on each instance, with timeouts set to 5 s for the 9×9 instances, 20 s for 16×16, and 120 s for 25×25. These timeouts are shorter than those used by [33]; however, we ran our experiments on a faster processor, and with compiler optimizations enabled. Taken together, these two differences should amount to a factor of approximately 3 in time. We designed the experiment so that each instance is used for one run; this is preferable to carrying out multiple runs on each of a smaller number of instances [37].

Figs. 4–6 show the results for average execution time (for successful runs) and success rate for the four algorithms. Summary results are given in Table II and the raw data are given in Table III. In Table III, we indicate in bold quantities which are significantly best of all algorithms, and with asterisks significant

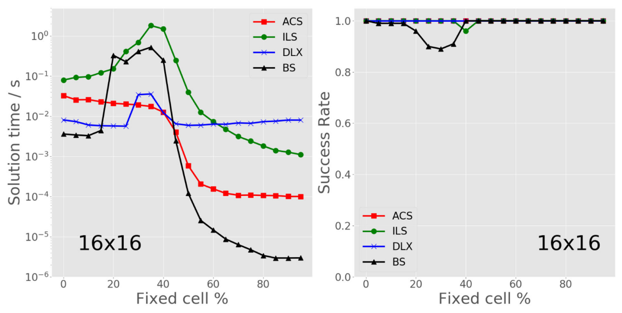


Fig. 5. Plots of solution time (left) and success rate (right) against fixed cell percentage for runs of ACS, ILS, DLX, and BS on the 16×16 general instances.

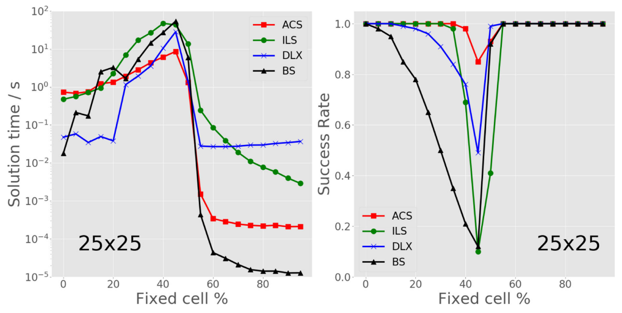


Fig. 6. Plots of solution time (left) and success rate (right) against fixed cell percentage for runs of ACS, ILS, DLX, and BS on the 25×25 general instances.

TABLE II
SUMMARY OF RESULTS ON THE GENERAL INSTANCES (20 FILLED CELL FRACTIONS FOR ORDERS 3, 4, AND 5)

		Algorithm 1				
	Algorithm 0	ACS	ILS	DLX	BS	All
Success	ACS	-	3	3	3	2
	ILS	0	-	0	0	0
	DLX	0	0	-	0	0
	BS	0	0	0	-	0
Time	ACS	-	52	37	11	2
	ILS	3	-	21	3	0
	DLX	21	38	-	9	5
	BS	48	52	45	-	40

The table shows the number of instance categories for which the algorithm listed in the first column (Algorithm 0) produces a significantly higher success rate, or lower mean solution time, than the other algorithms (Algorithm 1, or all algorithms). Significance is taken at the 0.05 level for pairwise comparisons, or $0.05/3$ for one-against-all comparisons. This data is summarized from Table III; details of the statistical tests are given in Section IV.

differences between the stochastic algorithms ACS and ILS. Statistical significance is tested using the χ^2 contingency test for the success rates, and the Mann–Whitney U test for the solution times. We use the Mann–Whitney test here as the vectors of times will in general have differing lengths. In cases where we test all algorithms against each other, we apply the Bonferroni correction to modify the p-value threshold for significance.

As in [14] and [33], we observe a "phase transition" in the difficulty of the instances as a function of the fixed cell fraction; the difficulty is markedly greater at fixed cell fractions of around 40%–50%. For low values of the fixed cell fraction, the search space is large, but there also exist many possible solutions. As the grid becomes denser, the size of search space decreases as well as the number of possible solutions. At around 45%, the combination of rarity of solutions and the size of the search space leads to a sharp peak in difficulty.

The most difficult puzzles are the 25×25 instances with a fixed cell fraction between 40% and 50%. For these fixed cell fractions of 40% and 45%, ACS outperforms the other three algorithms by a *significant* margin; ACS achieves success rates of 98% and 85% (compared to 69% and 10% for ILS, 76% and 49% for DLX, and 21% and 12% for BS). These are the only instances in all the experiments presented for which one algorithm achieved a significantly higher success rate than the other three. The mean times achieved by ACS on these instances are lower than the other three algorithms, but the difference is not statistically significant—this is most likely due to the small samples of times for the three algorithms which recorded low numbers of successes.

It is interesting to note the difference in performance between ACS and BS. These two codes use the same underlying problem representation and CP code; the only difference between them is the search strategy. This comparison is compelling evidence that ACS is very efficient at searching the solution space, giving markedly improved performance on the hardest instances over an exhaustive search strategy using the same underlying evaluation routines. For the easier instances, BS outperforms ACS, perhaps due to the simplicity of the algorithm which requires very little setup compared to ACS, or transformation to another problem representation, as in DLX.

ACS returns significantly lower runtimes than ILS, the other stochastic search algorithm, in 52 of the 60 instances, whereas ILS is significantly faster than ACS for only two instances. The performance of ACS on these general instances is significantly better than that of ILS both in terms of overall runtime, and success rate on the hardest instances.

D. Evaluation of Best Value Evaporation

In order to evaluate the effectiveness of BVE as an antistagnation mechanism, we ran experiments using the logic solvable instances (Section IV-B) and general instances (Section IV-C) using the ACS algorithm with BVE disabled by setting $\rho_{\text{BVE}} = 0$. We used all the logic-solvable instances, and the 25×25 general instances (since these are the most challenging). For the named 9×9 logic-solvable instances, we find that ACS without BVE performs very poorly on the harder instances (*aiescargot, coly013, goldennugget, platinumblond, reddwarf, tarx0134*), failing to solve these in most cases (see Table IV). Performance on the ten instances from [31] is similar to BVE, with the exception of *sabuncu6*, with a success rate of 95%. This suggests that these ten instances are not sufficiently difficult to provide a good benchmark for solution algorithms; the search space after applying constraints is either too small or, as is the case for four of the instances, nonexistent.

We also evaluated BVE using the general 25×25 instances. We see that the performance of ACS is *significantly degraded* without the BVE operator. Performance with respect to solution time is degraded to some extent, with significantly shorter times

TABLE III
SOLUTION RATES (SOLVED INSTANCES OUT OF 100) AND TIMES (MEAN AND STANDARD DEVIATION TIME OF SUCCESSFUL RUNS) FOR THE GENERAL INSTANCES

O	F(%)	Solution Rate				Solution Time/s			
		ACS	ILS	DLX	BS	ACS	ILS	DLX	BS
3	0	100	100	100	100	0.00187 ± 0.000653*	0.00778 ± 0.0022	0.00145 ± 0.000417	**0.000433 ± 0.000152**
3	5	100	100	100	100	0.00186 ± 0.000693*	0.00912 ± 0.00302	0.00132 ± 0.000391	**0.000372 ± 0.000138**
3	10	100	100	100	100	0.00156 ± 0.000563*	0.00974 ± 0.00443	0.00132 ± 0.000509	**0.000332 ± 0.000109**
3	15	100	100	100	100	0.00156 ± 0.000668*	0.0119 ± 0.00627	0.00117 ± 0.000373	**0.000300 ± 000101**
3	20	100	100	100	100	0.00142 ± 0.000569*	0.0152 ± 0.0101	0.00115 ± 0.00041	**0.000297 ± 0.000293**
3	25	100	100	100	100	0.00117 ± 0.000368*	0.0218 ± 0.0276	0.00107 ± 0.000338	**0.000219 ± 0.000131**
3	30	100	100	100	100	0.00091 ± 0.000354*	0.024 ± 0.0266	0.00107 ± 0.000332	**0.000226 ± 0.000325**
3	35	100	100	100	100	0.000572 ± 0.000256*	0.0202 ± 0.0243	0.0011 ± 0.000341	**0.000115 ± 0.0000878**
3	40	100	100	100	100	0.000266 ± 0.000207*	0.0102 ± 0.0178	0.00108 ± 0.000355	**$(4.52 \pm 3.7) \times 10^{-5}$**
3	45	100	100	100	100	0.000129 ± 0.000111*	0.00625 ± 0.0176	0.00103 ± 0.000349	**$(2.04 \pm 2.57) \times 10^{-5}$**
3	50	100	100	100	100	$(7.21 \pm 4.7) \times 10^{-5}$*	0.00176 ± 0.000971	0.00113 ± 0.000412	**$(1.03 \pm 1.06) \times 10^{-5}$**
3	55	100	100	100	100	$(5.54 \pm 4.98) \times 10^{-5}$*	0.0011 ± 0.000671	0.00114 ± 0.000384	**$(6.06 \pm 6.89) \times 10^{-6}$**
3	60	100	100	100	100	$(3.87 \pm 1.91) \times 10^{-5}$*	0.00094 ± 0.000526	0.00115 ± 0.000382	**$(3.27 \pm 3.68) \times 10^{-6}$**
3	65	100	100	100	100	$(3.5 \pm 1.78) \times 10^{-5}$*	0.00076 ± 0.000512	0.00122 ± 0.000441	**$(2.31 \pm 2.38) \times 10^{-6}$**
3	70	100	100	100	100	$(3.14 \pm 1.34) \times 10^{-5}$*	0.0006 ± 0.00051	0.00127 ± 0.000421	$(2.04 \pm 1.72) \times 10^{-6}$
3	75	100	100	100	100	$(3.41 \pm 1.72) \times 10^{-5}$	0.00056 ± 0.000571	0.00121 ± 0.000388	$(1.88 \pm 1.95) \times 10^{-6}$
3	80	100	100	100	100	$(3.24 \pm 1.19) \times 10^{-5}$	0.00051 ± 0.0005	0.00126 ± 0.000453	$(1.65 \pm 0.876) \times 10^{-6}$
3	85	100	100	100	100	$(3.43 \pm 1.31) \times 10^{-5}$*	0.0004 ± 0.00049	0.00127 ± 0.000451	**$(1.57 \pm 0.738) \times 10^{-6}$**
3	90	100	100	100	100	$(3.21 \pm 1.42) \times 10^{-5}$	0.00055 ± 0.000497	0.00146 ± 0.00052	$(1.58 \pm 0.681) \times 10^{-6}$
3	95	100	100	100	100	$(3.43 \pm 1.43) \times 10^{-5}$	0.00046 ± 0.000498	0.00145 ± 0.000511	$(1.49 \pm 0.700) \times 10^{-6}$
4	0	100	100	100	100	0.0327 ± 0.0129*	0.0802 ± 0.0205	0.0081 ± 0.00213	**0.00363 ± 0.00127**
4	5	100	100	100	99	0.0258 ± 0.0105*	0.0933 ± 0.0248	0.00739 ± 0.0042	**0.00342 ± 0.000962**
4	10	100	100	100	99	0.0261 ± 0.0103*	0.097 ± 0.0249	0.00608 ± 0.00167	**0.0033 ± 0.00186**
4	15	100	100	100	99	0.0231 ± 0.00846*	0.122 ± 0.0411	0.00583 ± 0.00158	**0.00441 ± 0.00742**
4	20	100	100	100	96	0.0212 ± 0.00958*	0.153 ± 0.0618	**0.00575 ± 0.0012**	0.331 ± 1.57
4	25	100	100	100	90	0.0202 ± 0.00885*	0.414 ± 0.945	**0.00567 ± 0.00178**	0.229 ± 1.87
4	30	100	100	100	89	**0.0191 ± 0.0111***	0.698 ± 1.3	0.0346 ± 0.287	0.414 ± 2.38
4	35	100	100	100	91	**0.0176 ± 0.0102***	1.84 ± 2.47	0.036 ± 0.254	0.525 ± 2.41
4	40	100	96	100	100	0.0127 ± 0.00945*	1.5 ± 1.98	0.0123 ± 0.0363	0.25 ± 1.27
4	45	100	100	100	100	0.00406 ± 0.00392*	0.246 ± 0.21	0.00652 ± 0.00165	**0.00248 ± 0.0076**
4	50	100	100	100	100	0.000588 ± 0.000564*	0.04 ± 0.0422	0.00596 ± 0.00192	**0.000121 ± 0.000194**
4	55	100	100	100	100	0.000205 ± 0.000162*	0.0125 ± 0.00548	0.00604 ± 0.00158	**$(2.55 \pm 2.57) \times 10^{-5}$**
4	60	100	100	100	100	0.000156 ± 0.000106*	0.00736 ± 0.00314	0.00638 ± 0.00196	**$(1.47 \pm 1.58) \times 10^{-5}$**
4	65	100	100	100	100	0.000121 ± 0.000059*	0.00476 ± 0.00224	0.00634 ± 0.00159	**$(8.72 \pm 9.37) \times 10^{-6}$**
4	70	100	100	100	100	0.000108 ± 0.000050*	0.00314 ± 0.00144	0.00684 ± 0.00171	**$(6.35 \pm 6.89) \times 10^{-6}$**
4	75	100	100	100	100	0.000109 ± 0.000050*	0.00241 ± 0.00102	0.0067 ± 0.00151	**$(4.73 \pm 5.52) \times 10^{-6}$**
4	80	100	100	100	100	0.000107 ± 0.000041*	0.00183 ± 0.000749	0.00734 ± 0.00162	**$(3.41 \pm 2.79) \times 10^{-6}$**
4	85	100	100	100	100	0.000105 ± 0.000044*	0.00141 ± 0.000634	0.0076 ± 0.0018	**$(2.94 \pm 1.49) \times 10^{-6}$**
4	90	100	100	100	100	0.000101 ± 0.000041*	0.00128 ± 0.000618	0.00805 ± 0.00208	**$(2.94 \pm 1.13) \times 10^{-6}$**
4	95	100	100	100	100	0.000101 ± 0.000033*	0.00111 ± 0.000488	0.00805 ± 0.0023	**$(2.96 \pm 1.55) \times 10^{-6}$**
5	0	100	100	100	100	0.731 ± 0.724	0.474 ± 0.0598	0.0476 ± 0.00759	**0.0178 ± 0.00351**
5	5	100	100	100	98	0.682 ± 0.661	0.561 ± 0.114*	0.0582 ± 0.173	0.213 ± 0.672
5	10	100	100	100	95	0.749 ± 0.931	0.715 ± 0.185*	0.0347 ± 0.0392	0.173 ± 0.791
5	15	100	100	99	85	1.23 ± 1.41	0.943 ± 0.671	0.0494 ± 0.17	2.54 ± 12.1
5	20	100	100	98	78	1.33 ± 1.53*	2.27 ± 2.68	0.0382 ± 0.0281	3.27 ± 13.6
5	25	100	100	96	65	1.93 ± 1.69*	7.0 ± 6.54	**1.16 ± 5.89**	1.69 ± 7.03
5	30	100	100	91	50	2.85 ± 2.52*	17.2 ± 8.8	**1.94 ± 11.1**	5.39 ± 16.5
5	35	100	98	84	35	4.36 ± 3.4*	26.7 ± 10.5	**3.57 ± 11.0**	14.5 ± 29.3
5	40	**98***	69	76	21	6.15 ± 5.61*	47.1 ± 25.7	10.5 ± 19.5	27.1 ± 37.3
5	45	**85***	10	49	12	8.61 ± 10.1*	43.4 ± 26.8	28.3 ± 37.4	53.8 ± 42.1
5	50	93*	41	99	92	1.3 ± 4.82*	13.6 ± 21.9	1.41 ± 5.73	6.01 ± 14.9
5	55	100	100	100	100	0.00152 ± 0.0082*	0.243 ± 0.284	0.0278 ± 0.00469	**0.000435 ± 0.00288**
5	60	100	100	100	100	0.000341 ± 0.000153*	0.0857 ± 0.0275	0.027 ± 0.00411	**$(4.36 \pm 3.19) \times 10^{-5}$**
5	65	100	100	100	100	0.000287 ± 0.000105*	0.039 ± 0.019	0.0269 ± 0.00405	**$(3.09 \pm 2.19) \times 10^{-5}$**
5	70	100	100	100	100	0.000246 ± 0.000080*	0.019 ± 0.00721	0.0276 ± 0.00416	**$(2.12 \pm 1.12) \times 10^{-5}$**
5	75	100	100	100	100	0.000227 ± 0.000063*	0.0109 ± 0.00381	0.0295 ± 0.00426	**$(1.55 \pm 0.596) \times 10^{-5}$**
5	80	100	100	100	100	0.000219 ± 0.000050*	0.0077 ± 0.00257	0.0298 ± 0.00407	**$(1.41 \pm 0.576) \times 10^{-5}$**
5	85	100	100	100	100	0.000228 ± 0.000061*	0.00584 ± 0.00194	0.0326 ± 0.00502	**$(1.42 \pm 0.494) \times 10^{-5}$**
5	90	100	100	100	100	0.00021 ± 0.000047*	0.00399 ± 0.00121	0.0343 ± 0.00527	**$(1.27 \pm 0.232) \times 10^{-5}$**
5	95	100	100	100	100	0.000212 ± 0.000051*	0.00289 ± 0.00103	0.0369 ± 0.00548	**$(1.28 \pm 0.309) \times 10^{-5}$**

O is the order of the puzzle (3 for 9×9, 4 for 16×16, 5 for 25×25) and F is the percentage of given cells. Figures in bold denote quantities for which one algorithm is significantly superior to the other three. For the solution times, the vectors of times are compared using the Mann-Whitney U test. Success rates are compared using a χ^2 contingency test. In all cases, the Bonferonni correction is applied, so that p values less than $0.05/3$ are taken to be significant. Asterisks show quantities for which either ILS or ACS are significantly superior to the other, using the same tests with $p < 0.05$.

TABLE IV
PERFORMANCE OF ACS WITH AND WITHOUT BVE ON THE SIXTEEN 9×9 LOGIC-SOLVABLE INSTANCES AND GENERAL 25×25 INSTANCES

	Success %		Solution Time/s	
Instance	BVE	No BVE	BVE	No BVE
sabuncu1	100	100	$(4.8e \pm 1.84) \times 10^{-5}$	$(4.55 \pm 1.85) \times 10^{-5}$
sabuncu2	100	100	$(4.82 \pm 1.73) \times 10^{-5}$	$(4.87 \pm 1.82) \times 10^{-5}$
sabuncu3	100	100	0.000993 ± 0.000457	0.000993 ± 0.000454
sabuncu4	100	100	0.000625 ± 0.000708	0.000563 ± 0.000277
sabuncu5	100	100	$(4.62 \pm 1.5) \times 10^{-5}$	$(4.8 \pm 1.7) \times 10^{-5}$
sabuncu6	**100**	93	$\mathbf{0.0107 \pm 0.00828}$	0.561 ± 0.976
sabuncu7	100	100	0.00106 ± 0.000986	0.00126 ± 0.0021
sabuncu8	100	100	0.000728 ± 0.000343	0.000713 ± 0.000334
sabuncu9	100	100	0.00163 ± 0.0014	0.00225 ± 0.00324
sabuncu10	100	100	$(4.73 \pm 1.85) \times 10^{-5}$	$(4.77 \pm 1.84) \times 10^{-5}$
aiescargot	**100**	59	$\mathbf{0.0204 \pm 0.0152}$	0.949 ± 1.36
coly013	**100**	24	0.0488 ± 0.0518	0.633 ± 1.14
goldennugget	**100**	34	$\mathbf{0.0374 \pm 0.0293}$	0.92 ± 1.15
platinumblond	**100**	7	$\mathbf{0.113 \pm 0.0859}$	1.05 ± 0.905
reddwarf	**100**	35	$\mathbf{0.0404 \pm 0.0354}$	1.64 ± 1.61
tarx0134	**100**	47	$\mathbf{0.0259 \pm 0.0193}$	1.5 ± 1.54
$25 \times 25\ 0\%$	100	100	0.731 ± 0.724	0.503 ± 0.347
$25 \times 25\ 5\%$	100	100	0.682 ± 0.661	0.469 ± 0.297
$25 \times 25\ 10\%$	100	100	0.749 ± 0.931	0.521 ± 0.363
$25 \times 25\ 15\%$	100	100	1.23 ± 1.41	$\mathbf{0.581 \pm 0.535}$
$25 \times 25\ 20\%$	100	100	1.33 ± 1.53	0.792 ± 0.594
$25 \times 25\ 25\%$	100	100	1.93 ± 1.69	$\mathbf{1.08 \pm 0.783}$
$25 \times 25\ 30\%$	100	100	2.85 ± 2.52	$\mathbf{1.77 \pm 1.3}$
$25 \times 25\ 35\%$	100	100	$\mathbf{4.36 \pm 3.4}$	4.59 ± 8.25
$25 \times 25\ 40\%$	**98**	83	6.15 ± 5.61	8.77 ± 15.6
$25 \times 25\ 45\%$	**85**	49	8.61 ± 10.1	9.14 ± 18.2
$25 \times 25\ 50\%$	**93**	80	1.3 ± 4.82	1.52 ± 11.0
$25 \times 25\ 55\%$	100	100	0.00152 ± 0.0082	0.00101 ± 0.00273
$25 \times 25\ 60\%$	100	100	$\mathbf{0.000341 \pm 0.000153}$	0.000411 ± 0.000208
$25 \times 25\ 65\%$	100	100	$\mathbf{0.000287 \pm 0.000105}$	0.000321 ± 0.000105
$25 \times 25\ 70\%$	100	100	$\mathbf{0.000246 \pm 0.000081}$	0.000274 ± 0.000075
$25 \times 25\ 75\%$	100	100	$\mathbf{0.000227 \pm 0.000063}$	0.000253 ± 0.000063
$25 \times 25\ 80\%$	100	100	$\mathbf{0.000219 \pm 0.000050}$	0.000243 ± 0.000055
$25 \times 25\ 85\%$	100	100	$\mathbf{0.000228 \pm 0.000061}$	0.000246 ± 0.000057
$25 \times 25\ 90\%$	100	100	$\mathbf{0.00021 \pm 0.000048}$	0.000239 ± 0.000049
$25 \times 25\ 95\%$	100	100	$\mathbf{0.000212 \pm 0.000051}$	0.000251 ± 0.000054

Success% is the number of successful solutions found in 100 runs. Times are given in seconds, with mean and standard deviation over 100 runs. Numbers in bold indicate statistically significant differences between the algorithms, determined using the Mann–Whitney U test for the times, and χ^2 contingency test for the success rates.

without BVE in three fixed cell fractions, compared to nine which are faster with BVE. The number of failures is significantly higher; for the 45% fixed cell instances for example, the success rate is 58%, compared to 92% with BVE enabled. The average solution time for these instances is 9.1 s, well within the timeout of 120 s, suggesting that the failures are due to the search stagnating at a local minimum.

V. CONCLUSION

In this article, we presented a new algorithm for the *Sudoku* puzzle, based on ACO. Our method includes a new operator, which we call *best value evaporation*, and we show that this addition to the base algorithm is essential for the prevention of premature convergence or stagnation of solutions. Experiments show that our new algorithm significantly outperforms existing algorithms on the hardest, large instances of *Sudoku*, and we provide evidence that our method provides a much more efficient search of the solution space than traditional backtracking algorithms for these problems. For smaller or easier instances, we find that direct search algorithms such as dancing links or BS outperform stochastic algorithms, but these deterministic algorithms perform poorly on the hardest instances. Finally, we find that our algorithm outperforms the state of the art ILS algorithm [33] both in terms of runtime and success rates on hard instances.

The growing body of work on the automated solution of pencil puzzles such as *Sudoku* and Nurikabe suggests that they offer a ready-made algorithmic test-bed. As such, they may provide an additional challenge for general-purpose algorithms (whether AI-based or not), and offer new insights into the solution of constraint satisfaction problems (by, for example, suggesting new ways in which to search the solution space).

Importantly, solvers such as ours can outperform state-of-the-art methods without any requirement for problem-specific heuristics, which immediately offers two possibilities for future work in this area. The first is a "problem agnostic" *general* Japanese pencil puzzle solver, which can solve large instances of any problem in this class. By constructing this solver in a modular fashion, we should easily be able to incorporate any suitable pencil puzzle, which will minimize the amount of effort required in future research. Importantly, this will allow for the rapid (and experimentally consistent) solution of a wide range of pencil puzzles, which will: 1) yield good solutions to these problems *per se*; 2) allow for easy comparison of the *properties* of those problems; and 3) provide a ready-made platform for the subsequent investigation of problem-specific heuristics.

REFERENCES

[1] J.-P. Delahaye, “The science behind Sudoku,” *Scientific Amer.*, vol. 294, no. 6, pp. 80–87, 2006.

[2] P. Norvig, “Solving every Sudoku puzzle,” Accessed: Mar. 13, 2018. [Online]. Available: http://norvig.com/sudoku.html

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: WH Freeman and Co., 1979.

[4] T. Yato and T. Seta, “Complexity and completeness of finding another solution and its application to puzzles,” *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. 86, no. 5, pp. 1052–1060, 2003.

[5] C. J. Colbourn, “The complexity of completing partial Latin squares,” *Discrete Appl. Math.*, vol. 8, no. 1, pp. 25–30, 1984.

[6] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. Berlin, Germany: Springer, 2018.

[7] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations*. Berlin, Germany: Springer, 1972, pp. 85–103.

[8] D. E. Knuth, “Dancing links,” *Preprint*, 2000, arXiv: cs/0011047.

[9] M. Hunt, C. Pong, and G. Tucker, “Difficulty-driven Sudoku puzzle generation,” *UMAP J.*, vol. 29, no. 3, pp. 343–361, 2007.

[10] S. Fletcher, F. Johnson, and D. R. Morrison, “Taking the mystery out of Sudoku difficulty: An Oracular model,” *UMAP J.*, vol. 29, no. 3, pp. 327–341, 2007.

[11] T. Weber, “A SAT-based Sudoku solver,” in *Proc. 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning: Short Paper Proceedings*, G. Sutcliffe and A. Voronkov, Eds., 2005, pp. 11–15.

[12] J. A. Pacurib, G. M. M. Seno, and J. P. T. Yusiong, “Solving Sudoku puzzles using improved artificial bee colony algorithm,” in *Proc. Fourth Int. Conf. Innovative Comput., Inf. Control*, IEEE, 2009, pp. 885–888.

[13] B. Crawford, M. Aranda, C. Castro, and E. Monfroy, “Using constraint programming to solve Sudoku puzzles,” in *Proc. Third Int. Conf. Convergence Hybrid Inf. Technol.*, IEEE, 2008, vol. 2, pp. 926–931.

[14] R. Lewis, “Metaheuristics can solve Sudoku puzzles,” *J. Heuristics*, vol. 13, no. 4, pp. 387–401, 2007.

[15] X. Q. Deng and Y. Da Li, “A novel hybrid genetic algorithm for solving Sudoku puzzles,” *Optim. Lett.*, vol. 7, no. 2, pp. 241–257, 2013.

[16] T. Mantere and J. Koljonen, “Solving, rating and generating Sudoku puzzles with GA,” in *Proc. IEEE Congr. Evol. Comput.*, 2007, pp. 1382–1389.

[17] C. Segura, S. I. V. Peña, S. B. Rionda, and A. H. Aguirre, “The importance of diversity in the application of evolutionary algorithms to the Sudoku problem,” in *Proc. IEEE Congr. Evol. Comput.*, 2016, pp. 919–926.

[18] Z. Wang, T. Yasuda, and K. Ohkura, “An evolutionary approach to Sudoku puzzles with filtered mutations,” in *Proc. IEEE Congr. Evol. Comput.*, 2015, pp. 1732–1737.

[19] J. M. Hereford and H. Gerlach, “Integer-valued particle swarm optimization applied to Sudoku puzzles,” in *Proc. IEEE Swarm Intell. Symp.*, 2008, pp. 1–7.

[20] A. Moraglio and J. Togelius, “Geometric particle swarm optimization for the Sudoku puzzle,” in *Proc. 9th Annu. Conf. Genetic Evol. Comput.*, ACM, 2007, pp. 118–125.

[21] Z. Karimi-Dehkordi, K. Zamanifar, A. Baraani-Dastjerdi, and N. Ghasem-Aghaee, “Sudoku using parallel simulated annealing,” in *Proc. Int. Conf. Swarm Intell.* Springer, 2010, pp. 461–467.

[22] R. Soto, B. Crawford, C. Galleguillos, E. Monfroy, and F. Paredes, “A hybrid ac3-tabu search algorithm for solving Sudoku puzzles,” *Expert Syst. With Appl.*, vol. 40, no. 15, pp. 5817–5821, 2013.

[23] J. Gunther and T. Moon, “Entropy minimization for solving Sudoku,” *IEEE Trans. Signal Process.*, vol. 60, no. 1, pp. 508–513, Jan. 2012.

[24] M. Dorigo and G. Di Caro, “Ant colony optimization: A new meta-heuristic,” in *Proc. Congr. Evol. Comput.*, IEEE, vol. 2, 1999, pp. 1470–1477.

[25] M. Dorigo, V. Maniezzo, and A. Colorni, “Ant system: Optimization by a colony of cooperating agents,” *IEEE Trans. Syst., Man, Cybern., B, Cybern.*, vol. 26, no. 1, pp. 29–41, Feb. 1996.

[26] M. Dorigo and M. Birattari, “Ant colony optimization,” in *Encyclopedia of Machine Learning*. Berlin, Germany: Springer, 2011, pp. 36–39.

[27] M. López-Ibáñez, T. Stützle, and M. Dorigo, “Ant colony optimization: A component-wise overview,” in *Handbook of Heuristics*. Cham, Switzerland: Springer International, 2016, pp. 1–37.

[28] M. Dorigo and T. Stützle, “Ant colony optimization: Overview and recent advances,” in *Handbook of Metaheuristics*. Berlin, Germany: Springer, 2019, pp. 311–351.

[29] T. Mantere, “Improved ant colony genetic algorithm hybrid for Sudoku solving,” in *Proc. Third World Congr. Inf. Commun. Technol.*, IEEE, 2013, pp. 274–279.

[30] K. Schiff, “An ant algorithm for the Sudoku problem,” *J. Autom., Mobile Robot. Intell. Syst.*, vol. 9, pp. 24–27, 2015.

[31] I. Sabuncu, “Work-in-progress: Solving Sudoku puzzles using hybrid ant colony optimization algorithm,” in *Proc. 1st Int. Conf. Ind. Netw. Intell. Syst.*, IEEE, 2015, pp. 181–184.

[32] A. Inkala, *AI Escargot - The Most Difficult Sudoku Puzzle*. Morrisville, North Carolina: Lulu.com, 2007.

[33] N. Musliu and F. Winter, “A hybrid approach for the Sudoku problem: Using constraint programming in iterated local search,” *IEEE Intell. Syst.*, vol. 32, no. 2, pp. 52–62, Mar.–Apr. 2017.

[34] M. Dorigo and L. M. Gambardella, “Ant colony system: A cooperative learning approach to the Traveling Salesman Problem,” *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 53–66, Apr. 1997.

[35] J. Laire, “dlx-cpp,” Accessed: Apr. 23, 2018. [Online]. Available : https://github.com/jlaire/dlx-cpp

[36] M. Ercsey-Ravasz and Z. Toroczkai, “The chaos within Sudoku,” *Sci. Rep.*, vol. 2, pp. 725–733, 2012.

[37] M. Birattari, “On the estimation of the expected performance of a metaheuristic on a class of instances. How many instances, how many runs?,” IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, Tech. Rep. TR/IRIDIA/2004-001, 2004.

Huw Lloyd (M’19) received the B.Sc. degree in physics from Imperial College, London, U.K. and the Ph.D. degree in astrophysics from the University of Manchester, Manchester, U.K.

He is a Senior Lecturer with Manchester Metropolitan University, Manchester, U.K.

Martyn Amos received the B.Sc. degree in computer science from Coventry University, Coventry, U.K. and the Ph.D. degree in DNA computation from the University of Warwick, Coventry, U.K.

He is a Professor of Computer and Information Sciences with Northumbria University, Newcastle upon Tyne, U.K.

Prof. Amos is a Fellow of the British Computer Society.