Metaheuristics can solve sudoku puzzles — **Source link** ↗

Rhyd Lewis

Institutions: Edinburgh Napier University

Related papers:

- Complexity and Completeness of Finding Another Solution and Its Application to Puzzles

- Sudoku as a Constraint Problem

- Solving, rating and generating Sudoku puzzles with GA

- Sudoku as a SAT Problem.

- Geometric particle swarm optimization for the sudoku puzzle

Share this paper: 🅕 🐦 in ✉

View more about this paper here: https://typeset.io/papers/metaheuristics-can-solve-sudoku-puzzles-frrcpfui0s

Metaheuristics can Solve Sudoku Puzzles.

Author: Rhyd Lewis
Affiliation: Centre for Emergent Computing,
Napier University, Scotland.

Correspondence Details:
Rhyd Lewis
School of Computing,
Napier University,
Edinburgh,
Scotland,
EH10 5DT.
Email: r.lewis@napier.ac.uk
Tel: 0131 455 2767.

Abstract: In this paper we present, to our knowledge, the first application of a metaheuristic technique to the very popular and *NP*-complete puzzle known as 'sudoku'. We see that this stochastic search-based algorithm, which uses simulated annealing, is able to complete logic-solvable puzzle-instances that feature daily in many of the UK's national newspapers. We also introduce a new method for producing sudoku problem instances (that are not necessarily logic-solvable) and use this together with the proposed SA algorithm to try and discover what types of instances this algorithm is best suited for. Consequently we notice the presence of an 'easy-hard-easy' style phase-transition similar to other problems encountered in operational research.

Keywords: Metaheuristics, Sudoku, Puzzles, Phase-Transition.

Introduction

Sudoku has been described as the 21st century's Rubik's cube (Pendlebury, 2005). Loosely meaning 'solitary number' in Japanese, sudoku is a popular and seemingly addictive puzzle that is currently taking many parts of the world by storm. In *The Independent*'s UK best-seller list for June 2005, for example, no fewer than three of the top ten spots were taken up with sudoku puzzle books.

The object of sudoku is simple: given an $n^2 \times n^2$ grid divided into $n \times n$ distinct squares (see figure 1), the aim is to fill each cell so that the following three criteria are met:

1. Each row of cells contains the integers 1 through to n^2 exactly once.
2. Each column of cells contains the integers 1 through to n^2 exactly once
3. Each $n \times n$ square contains the integers 1 through to n^2 exactly once.

In this paper, we will refer to the value of *n* as the puzzle's *order*.

The fundamental origins of sudoku actually lie within the work of the great 17th century Swiss mathematician Leonard Euler who, in 1783, reported on the idea of 'Latin Squares': grids of equal dimensions in which every symbol occurs exactly once in every row and every column. In fact, the problem of constructing Latin squares from a partially filled grid is *NP*-complete (Colbourn, Colbourn and Stinson, 1984), as is sudoku (Yato and Seta, 2003), which only differs to the Latin squares problem due to the presence of the third criterion above.

The sudoku puzzle first appeared in an American puzzle magazine under the name 'Number Place'. Later, the game also started to appear in Japanese puzzle magazines where it took its present name. Currently in Japan there are now five sudoku magazines published every month, with a total circulation of over 600,000 (Pendlebury, 2005).

Figure 1. An order-3 sudoku puzzle that can be easily solved through simple logic and reasoning, and which is typical of those found in many of the UK's daily newspapers.

Since being introduced into the UK by national newspaper *The Daily Mail*, the puzzle's popularity amongst British people has also rocketed. At the time of writing, multiple instances of the puzzle appear daily in many of the UK's most popular newspapers including *The Independent*, *The Times*, *The Sun*, as well as continuing in the *Daily Mail*. The vast majority of puzzle-instances found in these newspapers are order-3, although *The Independent* and *Irish Independent* for example, occasionally include puzzles of order-4 (that they call 'Super Sudoku'). With persistence, the experienced player can generally achieve completion of these larger grids and, if lucky, this can result in a prize. (Note that some publishers prefer to use numbers from zero through to n^2 - 1, or other characters in their specifications of what needs to go into the grid. However, this does not affect the puzzle in any way and, for clarity's sake, we will remain with our original definition.)

1 Making and Solving Sudoku Puzzles

As mentioned above, sudoku is an *NP*-complete problem (Yato and Seta, 2003). However, there are, in fact, a large number of grid configurations that can satisfy the three criteria of the game (see section 3.2 later, for example). In particular, we can easily a produce a valid solution when presented with a blank grid by simply applying the following algorithm (producing what we call the 'Root Solution').

> **ROOT-SOLUTION** (*n*)
> 1. $x \leftarrow 0$
> 2. **for** $i \leftarrow 1$ **to** *n* **do**
> 3.   **for** $j \leftarrow 1$ **to** *n* **do**
> 4.   **for** $k \leftarrow 1$ **to** n^2 **do**
> 5.   *grid* $[n(i$ - $1) + j][k] \leftarrow x$ (mod n^2) + 1
> 6. $x \leftarrow x + 1$
> 7. $x \leftarrow x + n$
> 8. $x \leftarrow x + 1$

To make the puzzle interesting and challenging, it is therefore typical for a number of the cells in the grid to be pre-filled by the puzzle master. The player's aim is to then complete the grid using the filled squares as guidance.

As an example of how a player might do this, consider the puzzle-instance given in figure 1. Here, we can deduce that the cell in the 7th row and 6th column (shaded) must be a 6, because all numbers 1 to 5 and 7 to 9 appear either in the same column, the same row, or the same square as this cell. If the problem-instance is designed well (as indeed this one is), the filling-in of this cell will present further clues, allowing the user to eventually complete the puzzle.

As can be imagined, the issue of designing good problem instances is particularly important if they are intended for human players. In general, puzzles should be *logic-solvable* – that is, they should admit exactly one solution, which is achievable through use of forward chaining logic only. Indeed, puzzles that force the player to make random choices (particularly when the grid is still quite empty) will quickly become tedious as he or she will have to go through the lengthy process of backtracking and re-guessing if previous guesses turn out to be wrong. On the other hand, puzzles that are too easy will also quickly become irksome.

Currently, order-3 and order-4 puzzles are made for UK newspaper *The Independent* by Mark Huckvale who uses an automated puzzle generator. According to Huckvale, a 'good' sudoku puzzle is

one that can be solved 'without guessing, in a logical sequence of steps from the initial configuration to a unique final solution' (Huckvale, 2005). The *Daily Mail*'s puzzles, on the other hand, are created by Peter Stirling who claims to do a lot of the puzzle generation by hand, although these are then checked for validity by a computer program (Pendlebury, 2005).

It may be concluded, therefore, that only certain sudoku problem instances actually make good puzzles. Consequently, puzzle creators need to take great care to ensure that their instances are enjoyable and satisfying to the player.

As well as solving sudoku puzzles by hand, there are also a number of specialised sudoku algorithms available on the web (e.g. at www.gwerdy.com and www.scanraid.com). Such algorithms typically mimic the logical processes that a human might follow, and standard logical rules, such as the so-called X-wing, and Swordfish rules (Armstrong, 2005), are also commonplace. Additionally, some of the more sophisticated algorithms, such as the 'Sudoku Solver by Logic' for order-3 puzzles (available at www.sudokusolver.co.uk) also incorporate simple backtracking schemes that can be used to undo guesses that turn out to be wrong (if, indeed, guesses are needed at all). Such algorithms, unsurprisingly, are often able to find solutions to the logic-solvable order-3 puzzles found in sudoku publications very quickly.

Lately, there have also been some sudoku-based papers published in the computer science literature: first by Simonis (2005), who formulates sudoku as a constraint satisfaction problem; and, more recently, by Lynce and Ouaknine (2006) who demonstrate how individual sudoku puzzles may be converted into conjunctive normal form, which can then be solved using propositional satisfiability techniques. In both of these studies, solutions to a large number of order-3, logic-solvable puzzles are quickly obtained by the authors' proposed algorithms, with no search being required. Higher order puzzles, however, are not considered in either case.

However, in all of the algorithms that we have considered so far, it is worth remembering that they are all, to some degree, dependant on being supplied with problem instances that have been specially constructed so that they definitely *can* be solved via the application of logical rules alone; not all problem instances, however, will have this property. Indeed, because, as we have mentioned, sudoku is an NP-complete problem, we know that we cannot hope to find a polynomial time algorithm for all possible problem instances, unless P = NP (Garey and Johnson, 1979). This implies that there will be some instances (possibly many) that cannot be solved without some sort of search also being necessary.

However, the systematic branch-and-bound style searches that are sometimes used as an add-on to thee sorts of algorithms are likely to slow up the run times in practice. As a simple demonstration, consider the order-3 problem-instance given in figure 1, with all of the 3s removed. In this instance, the first two algorithms (mentioned above) – which do not employ any search techniques – both actually halt with over thirty cells still unfilled. With regards to the more sophisticated 'Sudoku Solver by Logic', however, whilst this algorithm *is* able to solve this new problem-instance, it can only do so by making a number of guesses along the way. This, however, unfortunately results in a twenty-fold increase in its execution time.

Clearly then, the purely logic-based algorithms, whilst being ideal for very small and/or logic-solvable instances, may not actually be so suitable in other cases. Additionally, branch-and-bound searches are also likely become much more prohibitive for higher order problem instances, due to the much larger search spaces (and therefore potential timing implications) that will occur as a result. It is in these cases, therefore, that we might see a case for some sort of stochastic search being employed instead.

The remainder of the paper is now set out as follows: in section 2, we describe a new method for addressing sudoku problem instances that, instead of relying on logic-based rules, approaches this problem as one of optimisation. This is achieved via a simple representation, neighbourhood operator, and evaluation function, which we then use in conjunction with the standard simulated annealing metaheuristic (Kirkpatrick, Gelatt, and Vecchi, 1983). In section 3, we then conduct an analysis of this algorithm and also introduce a technique for generating problem instances. Finally, section 4 concludes the paper and provides a discussion on some future avenues of research.

2   A New Approach for Sudoku

For the remainder of this paper, the terminology that will be used is as follows: as before, a *square* refers to each $n \times n$ area of cells in the grid (denoted by the bold lines in figure 1). A square that is in row r and column c of squares will be denoted $square_{r, c}$. Secondly, the value contained in a cell in the ith row of cells and the jth column of cells is denoted as $cell_{i, j}$. If $cell_{i, j}$ has been filled with a specific value in the problem-instance, then it is described as *fixed* (in bold in figures 1 and 2), otherwise it is *non-fixed*. Finally, a complete grid that meets all three of the puzzle's criteria is referred to as *optimal*.

2.1 Representation and Neighbourhood Operator

Here, a direct representation is used. In order to form an initial candidate solution, we fill the grid by assigning each non-fixed cell in the grid a value. This is done randomly, but in such a way so that when the grid is full, every square contains the values 1 to n^2 exactly once.

During the run, the neighbourhood operator repeatedly chooses two different non-fixed cells in the same square and swaps them. In order to stop a sampling bias towards squares with less non-fixed values, this is done in the following way:

1. Choose randomly i and j, such that (a) $1 \leq i, j \leq n^2$, and (b) $cell_{i,j}$ is non-fixed.
2. Choose randomly k and l, such that (a) $cell_{k,l}$ is in the same square as $cell_{i,j}$, (b) $cell_{k,l}$ is non-fixed, and (c) $cell_{k,l} \neq cell_{i,j}$.
3. Swap $cell_{i,j}$ with $cell_{k,l}$.

Note that this method of producing an initial candidate solution, together with the defined neighbourhood operator, ensures that the third criterion of the puzzle is always met. The resultant search space size is therefore:

$$(1) \qquad \prod_{r=1}^{n} \prod_{c=1}^{n} f(r,c)!$$

(Where $f(r, c)$ is a function that indicates how many cells in $square_{r,c}$ are non-fixed).

2.2 Evaluation of Candidate Solutions

Because our representation and neighbourhood operator ensures that the third criterion of sudoku is always met, an appropriate cost function is obviously one that inspects for violations of the remaining two criteria. Our cost function looks at each row individually and calculates the number of values, 1 through to n^2 that are *not* present. The same is then done for each column, and the cost is simply the total of these values. (An example is provided in figure 2). Obviously, an optimal solution will have a cost of zero.

In this approach, we also use the process of delta-evaluation (Ross, Corne, and Fang, 1994) - note that a single application of the above neighbourhood operator means that *at most* two rows of cells and two columns of cells are affected by a swap. Thus, after a single move in the neighbourhood, rather than re-evaluate the whole candidate solution, only the parts that have been altered are recalculated. Such a scheme offers a considerable speed-up to the algorithm, particularly when considering problems of a higher order.
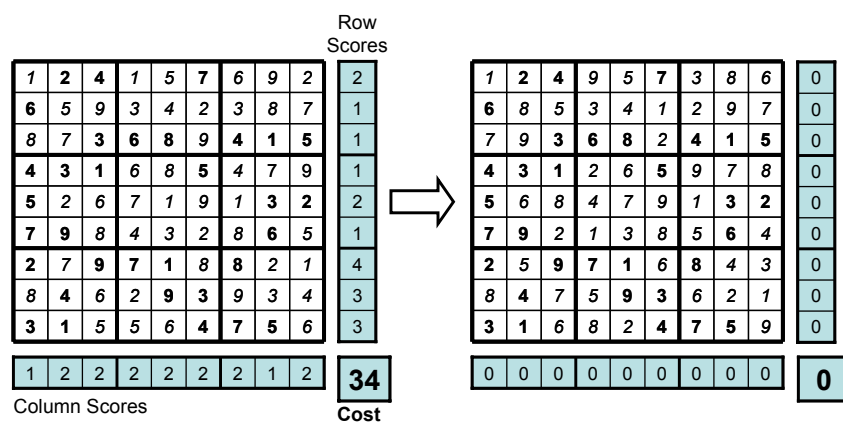

Figure 2. An example of how an initial candidate solution might be constructed using the problem-instance in figure 1, together with the resultant cost (left). Also shown is an optimal solution to this problem-instance (right).

2.3 A Simulated Annealing Algorithm

Having defined a suitable representation, neighbourhood operator, and evaluation function, the application of the simulated annealing (SA) metaheuristic is now straightforward.

Starting with an initial candidate solution, an exploration of the search space is conducted by iteratively applying the above neighbourhood operator. Given a candidate solution s, a neighbour s' is then accepted if (a) s' is better than s (with respect to the cost function), or (b) with a probability:

$$(2) \qquad\qquad \exp(-\delta/t)$$

where δ is the proposed change in the cost function and t is a control parameter, known as the temperature. Moves that meet neither of the above two conditions are reset.

In general, the way in which t is altered during the run is of great importance to the success and expense of the SA algorithm. It is generally believed that the initial temperature t_0 should allow the majority of moves to be accepted (van Laarhoven and Aarts, 1987), and then, during the run, it should be slowly reduced (according to a *cooling schedule*) so that the algorithm becomes increasingly greedy. If the algorithm is successful, eventually the search should converge at (or near to) a global optimum.

However, t_0 should be chosen with care: a value that is too high will mean that computation time is wasted because the algorithm is likely to conduct an unhelpful random walk through the search space. On the other hand, a value for t_0 that is too low will also have a negative impact, as it will make the search too greedy from the outset, and therefore make it more susceptible to getting stuck in local minima.

Van Laarhoven and Aarts (1987) advise that t_0 should allow approximately 80% of proposed moves to be accepted. One method for calculating this can be achieved by measuring the variance in cost for a small sample of neighbourhood moves. This scheme is actually based on the real-world process of annealing in metallurgy, which is beyond the scope of this paper, but may be found in the aforementioned text. We use this approach here - before starting the SA part of the algorithm, a small number of neighbourhood moves are performed. The initial temperature t_0 is simply the standard deviation of the cost during these moves.

During the run, the temperature is reduced using a simple geometric cooling schedule where the current temperature t_i is modified to a new temperature t_{i+1} via the formula:

$$(3) \qquad\qquad t_{i+1} = \alpha \cdot t_i$$

where α is a control parameter known as the *cooling rate*, and $0 < \alpha < 1$. Obviously, a large value for α, such as 0.999, will cause the temperature to drop very slowly, whilst a setting of, say, 0.5 will cause a much quicker cooling.

In this approach we also use the version of SA known as the *homogenous* SA scheme. This basically means that the algorithm takes the form of a sequence of Markov Chains, where each Markov chain is generated at a fixed value of t, and t is then altered in-between subsequent chains (van Laarhoven and Aarts, 1987). Again, the value defining the length of each Markov chain (here we call this value *ml*) also needs to be chosen with care: chains that are too short may not provide a thorough enough exploration of the search space, whilst chains that are too long will cause excessive run times. As with other practical applications of the SA algorithm (e.g. Abramson, Krishnamoorthy, and Dang, 1996), we decided that *ml* should be representative of the size of the problem-instance. Because this is defined by the number of non-fixed cells in the grid, *ml* is therefore calculated as follows:

$$(4) \qquad\qquad ml = \left(\sum_{r=1}^{n} \sum_{c=1}^{n} f(r,c) \right)^2$$

where $f(r, c)$ has the same meaning as section 2.1. Note that such a value allows a good chance of each pair of non-fixed cells being considered at least once per Markov chain.

Finally, we add a random-restart mechanism to the algorithm: If no improvement in cost is made for a fixed number of Markov chains (in all tests this was set to 20), then t is reset to its initial setting t_0, a new initial solution is generated, and the algorithm then continues as normal. We term this process a *reheat*.

3 Experimental Analysis

3.1 Solving Published Puzzles

Our first set of tests involved trying out this algorithm on a large number of order-3 sudoku puzzles (of various degrees of supposed 'difficulty') found in the UK's national newspapers, and also every super sudoku instance (order-4) that appeared in *The Independent* during June 2005.

Using a cooling rate $\alpha = 0.99$ and running all experiments on a PC under Linux (2.66Ghz processor and 1GB of RAM), we found that all order-3 puzzles were quickly solved by this algorithm, regardless of what difficulty rating the newspaper had given it. Typically, this was done in around half a second with no reheat being required. It is worth mentioning that although these puzzles might seem small, the search spaces are actually still fairly large (e.g. using our representation, the order-3 puzzle in figure 1 will have approximately 4.4937×10^{21} possible configurations), and so we believe that these results are pleasing, especially considering that the instances, in general, are deliberately constructed to admit only one optimal solution (Huckvale, 2005).

For the order-4 puzzles, our algorithm also found solutions in every case. However, this generally took a little longer to achieve (between 5 and 15 seconds), as sometimes one or two reheats were required. An example run with one of these order-4 instances can be viewed in figure 3.
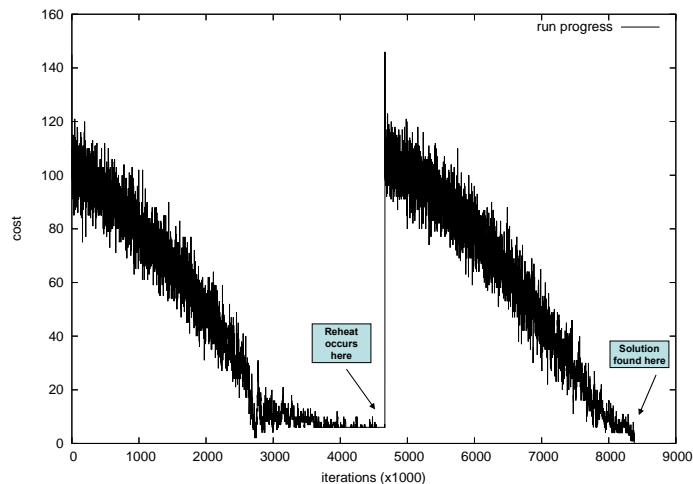

Figure 3. A typical run of the SA algorithm using an order-4 instance taken from *The Independent* on Saturday, June 18th 2005. This run required one reheat and took approximately 8 seconds to find a solution.

It must be noted straightaway that better run times for the order-3 puzzles (and, where appropriate, the order-4 puzzles) are often displayed by the logic based algorithms mentioned in section 1. However, given the fact that these particular puzzles are specifically designed so that they are solvable by the very methods employed by these algorithms, this is hardly surprising. This SA algorithm, on the other hand, was able to find solutions to these instances without having to make any assumptions on the initial configurations.

3.2 Creating Solvable Problem instances

Having seen that this SA algorithm is able to consistently solve published sudoku puzzles, and also noted that, in order to be successful, it does not necessarily require a sudoku problem-instance to be logic-solvable (i.e. specially constructed to be for solving via forward chaining logic alone), the question that now presents itself is: What actually *does* make a problem-instance difficult to solve with this particular algorithm?

In an attempt to empirically answer this question, we decided to produce our own instance generator. This, we achieved, by implementing a simple algorithm whereby complete, optimal grids are first obtained, and then some of the cells are removed from each one, according to some criteria (see below).

First, in order to produce large numbers of optimal grids, we make use of the fact that, given one optimal solution, we can alter it by:

- permuting columns of squares (n! possible permutations);

- permuting rows of squares ($n!$ possible permutations);
- permuting columns of cells within a single column of squares ($n!^n$ possible permutations); and
- permuting rows of cells within a single row of squares ($n!^n$ possible permutations),

and its optimality will still hold. Thus, given one optimal sudoku solution, there are in fact:

$$(5) \qquad\qquad\qquad\qquad n!^{2(n+1)} - 1$$

other optimal solutions obtainable from it.

   (Note that we are also able to take any optimal solution and switch the numbers (i.e. change all the 1s to, say, 4s, then change all the 4s to, say, 2s, and so on), thus providing a further ($n!$ - 1) possible grids obtainable from any one. However, whether this actually changes the puzzle in any meaningful sense, or just simply makes it look different to the human eye, is open to debate. For our purposes we decided not to include this additional procedure in our instance generator.)

   Moving our attention towards the issue of removing cells from one of these optimal grids, we note that if the problem-instance were intended to be a good puzzle for human players, then we would need to ensure that the logic-solvable property were always held. In practice, this can easily be achieved via an iterative scheme whereby each cell in the grid is considered in turn (possibly in a random order) and is then made blank if and only if its value can also be deduced (in some way) by the remaining filled cells in the grid. (Some useful discussion on this can also be found in the work of Simonis (2005).

   However, we note that the utilisation of a scheme such as this would actually impose severe limitations on what problem instances could be achieved; not least because in practice we are only able to remove a certain proportion of cells before the logic-solvable property is lost. (With order-3 puzzles, for example, a consultation of published sudoku puzzles, as well as our own experiences, suggests this to be approximately 70%)

   Thus, in an effort to look at the problem in more general terms (and to also try and answer the question posed at the beginning of this subsection), we therefore chose to use a more unrestricted method of cell removal. Our scheme therefore operates as follows: given an optimal solution, we define a parameter p (where $0 \leq p \leq 1$). Each cell in the grid is now considered in turn, and is removed with a probability of exactly $1 - p$. Obviously, this means that low values for p will tend to give fairly unconstrained problem instances (i.e. grids with a low proportion of fixed cells), whilst larger values for p will usually give more highly constrained problem instances (i.e. grids with a larger proportions of fixed cells).

   Note that it is quite possible, particularly for lower values of p, that this method of cell removal could produce problem instances that have more than one optimal solution. Note also that this method of cell removal, by paying no heed to the logic-solvable property, is more than likely to be inappropriate if the problem instances are intended to make good puzzles for humans to solve by hand.

3.3 Locating Phase-transitions

Using our instance generator, we performed a large number of experiments: for values of p from 0 through to 1.0 (incrementing in steps of 0.05), we first produced twenty different problem instances. With each instance, we then performed twenty trials using our SA algorithm. This we did for instances of order-3 (81 cells), order-4 (256 cells), and order-5 (625 cells), using time limits of 5, 30 and 350 seconds respectively.

   We noted in section 3.2 that our instance generator first needs to be seeded with a full and optimal solution (which can then be shuffled using the four listed operators). We obtained these grids from three sources: firstly, by taking some solution grids from *The Independent*; secondly, by executing our SA algorithm, using a blank grid as the problem-instance; and third, by using the puzzle's root solution (generated using the algorithm given in section 1). Note, however, that because, to our knowledge, there are no current sudoku publications that make use of order-5 puzzles, in this case we only used the latter two methods for producing optimal grids.

   Also, in our experiments, similarly to Eiben, van der Hauw, and van Hemert (1998) and Ross, Corne, and Terashima-Marin (1996), we choose to monitor algorithmic performance using two measurements: *success rate* and *solution time*. The success rate indicates the percentage of runs where a solution was found within the specified time limits (note that the maximum rate achievable is always 100%), and the solution time indicates the average amount of time that it took to find a solution. If the success rate was less than 100%, then those runs where optimality was not found within the time limit were not considered in the latter's calculation.

Figures 4, 5, and 6 show the results of the experiments where we used our SA algorithm to produce the seed for the instance generator. Note, however, that equivalent results (not shown here) were also achieved using the other two methods.
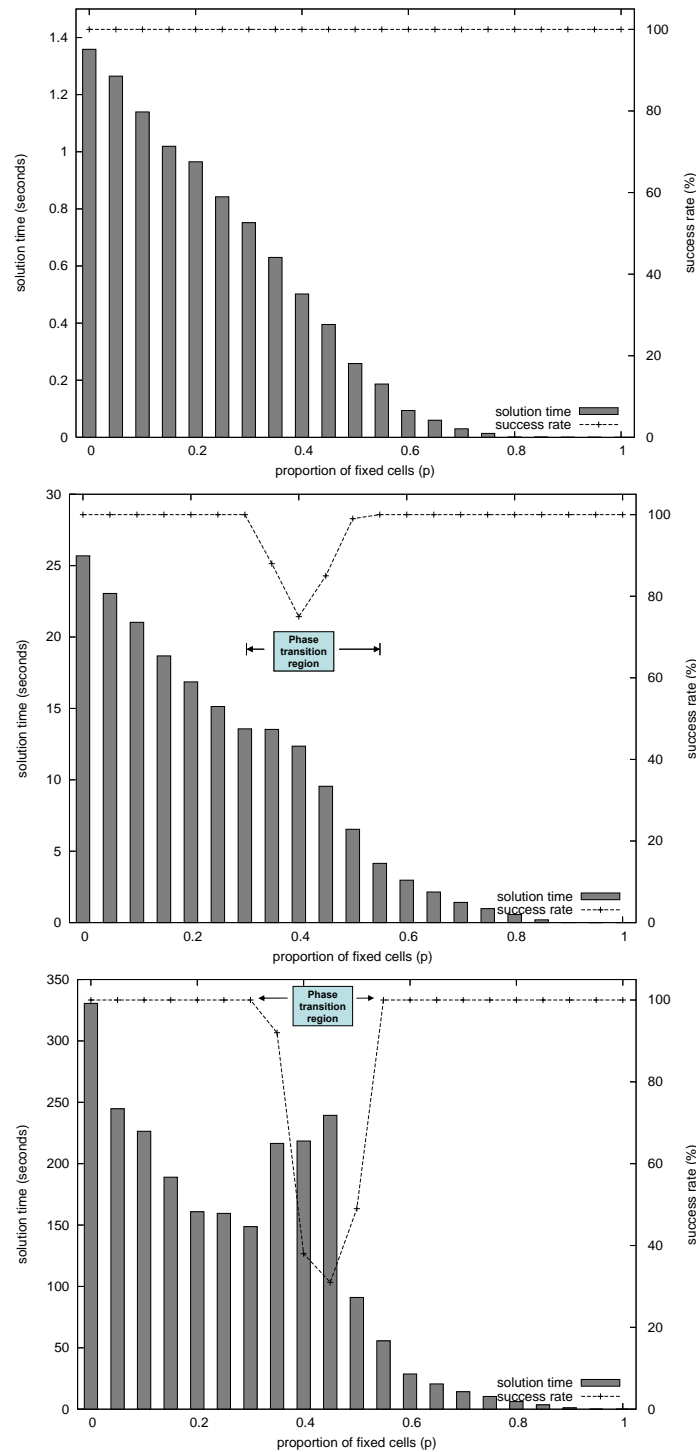
Figures 4, 5, and 6. The algorithm's performance with order-3, order-4 and order-5 problem instances (top, middle, and bottom respectively) for varying values of p, using a cooling rate of $\alpha = 0.99$.

Drawing our attention to figure 4 first (where $n = 3$), two things can be noticed. Firstly, we see that when the proportion of fixed cells is low (i.e. a low value for p) it generally takes longer to find an optimal solution. This is intuitive as, according to equation (1), emptier grids present a larger search space for the algorithm to navigate. Additionally, we note that in our algorithm the length of the Markov chains is also determined by the number of fixed cells in the grid (equation (4)), which also

results in longer runs (due to a slower cooling). The second thing to notice, however, is that the algorithm shows a 100% success rate for all of the trials.

Considering figure 5 next (where $n = 4$), we now see some new features emerging. As before, when the proportion of fixed cells is increased, the average amount of time to find a solution decreases. However, in this case we also see a dip in the success rate of the algorithm between values $p = 0.3$ and 0.55. Also around this point we see a very slight fluctuation in the solution times.

Finally, if we move our attention to figure 6 (where $n = 5$) we now see these features becoming much more prominent, and the characteristics of a phase-transition become clear: here, between the same values of p as figure 5, we see a much sharper drop in the success rate (reaching as low as 30% at $p = 0.45$) and also a large fluctuation in the solution time at the same points.

It is worth noting that 'easy-hard-easy' phase-transitions such as this are common in many *NP*-complete problems encountered in operational research (Cheeseman, Kanefsky and Taylor, 1991; Ross, Corne, and Terashima-Marin, 1996; Smith, 1994; Turner 1988). We now consider why such a transition might occur in this case:

When the proportion of fixed cells p is very low (or, in other words, the problem-instance has few constraints), there are, of course, many optimal solutions present within the search space, meaning that the search will generally be successful. However, according to equations (1) and (4), this also means that the search space is larger and the Markov chains are longer. In practice, this means that while the algorithm generally finds a solution without any reheating being necessary, the solution times are generally greater.

At the other extreme, when the proportion of fixed cells is large (or, in other words, the problem-instance is highly constrained), the search space and Markov chain lengths are reduced in size. Additionally, although in these cases there is probably only a very small number of optimal solutions within the search space, the very presence of so many fixed cells is likely to mean that the optimal solution lies in a deep local minimum (strong basin of attraction) allowing it to be easily found. These two factors therefore allow the algorithm to give low solution times and high success rates. (This second point also agrees with Ross, Corne, and Terashima-Marin (1996) who claim, with regards to their *NP*-complete timetabling problem, that as the number of constraints is increased, the sizes of the fitter plateaus in the fitness landscape are reduced.)

The phase-transition region lies in between these two extremes. Here, two characteristics ensure that the problem instances are more difficult to solve with this algorithm: firstly, we have a moderately large search space that only admits a small number of optimal solutions; secondly, we have a fitness landscape that possesses a large number of plateaus and local minima (Cheeseman, Kanefsky and Taylor, 1991). These features can be viewed most noticeably in figure 6, where a vast drop in the success rate can be witnessed for values of p between 0.3 and 0.55. Lastly, we note that when optimal solutions *were* found for problem instances within this region, because the search was more of a 'needle in a haystack', the algorithm often only did so after performing a number of reheats. The consequences of this can also be seen in figure 6 where the solution times in the phase-transition region are significantly higher than the solution times immediately to their left.

4   Conclusions and Discussion

In this paper, we have presented, to our knowledge, the first application of a metaheuristic technique to the popular sudoku puzzle. When applying this algorithm to a large number of problem instances taken from the UK press (of various degrees of 'difficulty'), we have seen in all our tests that this algorithm does not just get close to optimality (as is often the case with optimisation techniques), but consistently finds the solution in reasonably short amounts of time. We have also demonstrated that this method, particularly for lower order puzzles, is perhaps more robust than many existing algorithms, because in order to be successful, it does not necessarily depend on problem instances being logic-solvable.

Through a large number of experiments with our own instance generator, in this study we have also witnessed the existence of easy-hard-easy style phase-transition in larger puzzles (order 4 and 5), which are similar to those found in other NP-complete problems such as constraint satisfaction problems (Smith, 1994), timetabling problems (Ross, Corne, and Terashima-Marin, 1996), and graph colouring problems (Turner, 1988). Our experiments have indicated that in these cases, problem instances with around a 0.4 to 0.45 proportion of fixed cells seem to present this algorithm with the most difficulty, and that the effects of the phase-transition seem to become more pronounced as the order of the puzzle is increased. These observations are useful because, although this algorithm does not require a puzzle to be logic-solvable for it to be successful, it is still important to know where the methodology's general limitations lie. Note, however, that for order-3 puzzles, we do not see such a phase transition and, instead, witness a 100% success rate in all cases.

With regards to improving our algorithm, it is worth noting that there are a number of ways that this could be achieved: we could, for example, try to employ a more sophisticated cooling schedule such as those presented by Abramson, Krishnamoorthy, and Dang (1996) for the school timetabling problem. It is also quite possible that this representation, neighbourhood operator, and cost function could be used with other metaheuristics that are similar to SA such as tabu search (Glover and Laguna, 2002) or iterated local-search (Lourenco, Martin, and Stützle, 2002).

Finally, although we have demonstrated in this paper that this sort of stochastic search approach is able to solve a large variety of different instances unaided, perhaps the most salient point arising from this research on the whole is the obvious potential for combining both these techniques *and* logical-based approaches (such as those reviewed in section 1). Indeed, although we have often contrasted these two basic methodologies in this study, it is worth noting that the relationship between these two approaches may well in fact be symbiotic; that is, they may well be able to profit from one another's strengths: a stochastic search algorithm could help a logic-based algorithm to solve a wider range of problem instances, and logic-based algorithms have the potential drastically reduce/prune the search space in order to decrease the work of a stochastic search algorithm. Thus, a hybrid algorithm for sudoku (and other related problems) that, for example, takes a problem instance and follows the methodology of (a) filling as many cells as possible via logical rules (and thus hopefully moving it away from the phase transition region), and then (b) only switching to a stochastic search technique when guesses are finally required (if at all); has the potential to provide a much more powerful algorithm than either of these two techniques individually. It is also conceivable that other possible combinations might also show even further promise, although all of this is obviously subject to future research.

Acknowledgments

References

Abramson, D., H. Krishnamoorthy, and H. Dang. (1996). 'Simulated Annealing Cooling Schedules for the School Timetabling Problem.' *Asia-Pacific Journal of Operational Research* 16, 1 – 22.

Armstrong, S. (2005). X-wing and Swordfish. Online Resource, Accessed July 2005, http://www.simes.clara.co.uk/programs/sudokutechnique6.htm

Cheeseman, P., B. Kanefsky and W. Taylor. (1991). 'Where the Really Hard Problems Are' In *Proceedings of IJCAI-91*, 331 – 337.

Colbourn, C. J., M. J. Colbourn, and D. R. Stinson. (1984). 'The Computational Complexity of Recognising Critical Sets.' Graph Theory, Springer LNM 1073, 248 – 253.

Eiben, A. E., J. van der Hauw, and J. van Hemert. (1998). 'Graph Coloring with Adaptive Evolutionary Algorithms' *Journal of Heuristics* 4, 1, 25 – 46.

Garey, M. R. and D. S. Johnson. (1979). Computers and Intractability. W. H. Freeman, San Francisco.

Glover, F. and M. Laguna. (2002). 'Tabu search.' In P. M. Pardalos and M. G. C. Resende (eds.) Handbook of Applied Optimization, Oxford University Press, 194 – 208.

Huckvale, M. (2005). Mark Huckvale – Sudoku Puzzles. Online Resource accessed July 2005. http://www.phon.ucl.ac.uk/home/mark/sudoku/

Kirkpatrick, S., C. Gelatt, and M. Vecchi. (1983). 'Optimization by Simulated Annealing.' *Science* 4598, 671- 680.

Lourenco, H. R., O. Martin, and T. Stützle. (2002). 'Iterated Local Search.' In F. Glover and G. Kochenberger, (eds.) Handbook of Metaheuristics. Norwel, MA: Kluwer Academic Publishers. 321 – 353

Lynce, I. and Ouaknine. (2006). 'Sudoku as a SAT problem' In proceedings of the 9th Symposium on Artificial Intelligence and Mathematics, 2006.

Pendlebury, P. (2005). 'Can you Sudoku'. Article appearing in *The Mail on Sunday*, London, 8th May 2005. An online version is also available at the following address (accessed July 2005) http://www.mailonsunday.co.uk/pages/live/articles/news/news.html?in_article_id=348348&in_page_id =1770&in_a_source=

Ross, P, D. Corne, and H-L. Fang. (1994). 'Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation.' In Y. Davidor, H. Schwefel, M. Reinhard (eds.) Parallel Problem Solving From Nature III (PPSN) Springer-Verlag LNCS 866, 556 – 565.

Ross, P., D. Corne, and H. Terashima-Marin. (1996). 'The Phase-Transition Niche for Evolutionary Algorithms in Timetabling.' In Edmund Burke and Peter Ross (eds.) The Practice and Theory of Automated Timetabling (PATAT), Springer-Verlag LNCS 1153, 309 – 325.

Simonis, H. (2005). 'Sudoku as a Constraint Problem'. In *CP Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 13-27, Spain, October 2005.

Smith, B. (1994) 'Phase Transitions and the Mushy Region in Constraint Satisfaction Problems'. In A. Cohn (ed) Proceedings of the 11th European Conference on Artificial Intelligence, John Wiley and Sons Ltd, 100 – 104.

Turner, J. S. (1988). 'Almost all k-Colorable Graphs are Easy to Color.' *Journal of Algorithms*, 9, 63 – 82.

van Laarhoven, P, E. Aarts. (1987). Simulated Annealing: Theory and Applications. D. Reidel Publishing Company, The Netherlands.

Yato, T. and T. Seta. (2003). 'Complexity and Completeness of Finding Another Solution and Its Application to Puzzles'. *IEICE Trans. Fundamentals*, Vol. E86-A, No. 5, pp. 1052-1060.