Construction of Heuristics for a Search-Based Approach to Solving Sudoku

Conference Paper · January 2007

DOI: 10.1007/978-1-84800-094-0_4 · Source: DBLP

3 authors:

Sian K. Jones
University of South Wales
13 PUBLICATIONS **38** CITATIONS
SEE PROFILE

Paul Alun Roach
University of South Wales
49 PUBLICATIONS **458** CITATIONS
SEE PROFILE

Stephanie Perkins
University of South Wales
40 PUBLICATIONS **197** CITATIONS
SEE PROFILE

Construction of Heuristics for a Search-Based Approach to Solving Sudoku

S. K. Jones, P. A. Roach, S. Perkins
Department of Computing and Mathematical Sciences, University of Glamorgan, Pontypridd, CF37 1DL, United Kingdom, skjones@glam.ac.uk

Abstract

Sudoku is a logic puzzle, consisting of a 9×9 grid and further subdivided into 'mini-grids' of size 3×3. Each row, column, and 3×3 mini-grid contains the numbers 1 to 9 once, with a true Sudoku grid having a unique solution. Sudoku, along with similar combinatorial structures, has relationships with a range of real-world problems. Much published work on the solution of Sudoku puzzles has acknowledged the link between Sudoku and Latin Squares, thereby recognising the scale of any search space of possible solutions and that the generalization of the puzzle to larger grid sizes is NP-complete. However, most published approaches to the solution of Sudoku puzzles have focussed on the use of constraint satisfaction algorithms that effectively mimic solution by hand, rather than directly exploiting features of the problem domain to reduce the size of the search space and constructing appropriate heuristics for the application of search techniques. This paper highlights important features of the search space to arrive at heuristics employed in a modified steepest ascent hill-climbing algorithm, and proposes a problem initialization and neighbourhood that greatly speed solution through a reduction of problem search space. Results shown demonstrate that this approach is sufficient to solve even the most complex rated puzzles, requiring relatively few moves. An analysis of the nature of the problem search space is offered.

1 Introduction

Sudoku is a number-based logic puzzle, consisting of a 9×9 grid and further subdivided into 'mini-grids' of size 3×3. Each row, column, and 3×3 mini-grid contains the numbers 1 to 9 once, with a true Sudoku grid having a unique solution. Seemingly based primarily on Latin Squares and Magic numbers [1], Sudoku is a puzzle that has achieved great popularity relatively recently. It is attributed to Howard Garns and first published by Dell Magazines in 1979 [2]. Initially, the puzzle was called *Number Place*, but after gaining popularity in Japan it was trademarked by the Nikoli puzzle group under the name Sudoku (*Su* meaning number, and *Doku* meaning single) [2]. The puzzle became internationally successful from about 2005, and has sparked a trend of similar puzzles of different sizes and border constraints. Through its strong relationships with Latin Squares and other combinatorial structures, Sudoku is linked to real-world applications,

including conflict free wavelength routing in wide band optical networks, statistical design and error correcting codes [3], as well as timetabling and experimental design [4].

A Sudoku puzzle includes a partial assignment of the values. These assignments are commonly known as *givens* or predefined cells, and are unmovable. Sufficient givens are provided so as to specify a unique solution to the puzzle (*i.e.* the puzzle is *well-formed*). The givens should be chosen so that no given is itself a "logical consequence" [5] of the other givens (*i.e.* no redundant clues are provided). The objective is then to complete the assignment by inserting the missing values in such a way as to satisfy the constraints, by hand using logic or, as is increasingly the trend, by using *automated solutions*. No general means is yet known for determining the minimum number of givens required for a partial assignment that leads to a unique solution [6]. Much experimental work has been conducted in this area, however, and a well-formed puzzle has been constructed with just 17 givens [7]; this is the smallest number achieved to date.

Puzzles are typically categorised in terms of the difficulty in completing them by hand, with the use of four or five rating levels being common. These rating levels are subjective, meaning that the actual ranges and labels can vary greatly. Terms such as 'easy', 'medium' and 'hard' (and even 'tough' and 'diabolical') are commonly used. To categorize the complexity of a published puzzle, standard constraint based solvers (described below) may be used, and some publications use measures of the time taken by groups of human solvers to arrive at the solution. The complexity of a puzzle does not have a simple relationship with the number of givens – the positioning of the givens is a more important determinant of their value as 'hints' than their quantity.

Automated solutions could reasonably be divided into two categories; *constraint based approaches* (some of which effectively mimic the methods that one would use when solving the problem by hand), and *heuristic search optimization algorithms* (which turn some or all problem constraints into aspects of an optimization evaluation function). The latter category is the focus of this paper. Such approaches directly exploit features of the problem domain in order to reduce time spent examining a search space (the space of all possible solutions to given problem) [8], and are expected to be more generalizable to the solution of puzzles of larger grid sizes, where solution becomes more difficult.

The purpose of this paper is to highlight important features of the search space for Sudoku, to arrive at suitable heuristics for effective solution of a range of Sudoku puzzles. Through the testing of a modified steepest ascent hill-climbing algorithm, an effective problem initialization and neighbourhood definition are described that greatly speed solution by reducing the problem search space. Finally, an analysis of the nature of the problem search space, and of the appropriateness of heuristic search for this problem, is offered.

2 Literature Survey

Many programs for the automatic solution of Sudoku puzzles are currently commercially available. Although generally undocumented, these automatic solvers seem to rely on the types of solving rules used when solving by hand, combined

with a certain amount of trial and error [9]. Computer solvers can estimate the difficulty for a human to find the solution, based on the complexity of the solving techniques required.

Several authors have presented Sudoku as a constraint satisfaction problem [5, 10, 11]. In these approaches, rather than attempting to employ an objective function to determine a path through a state space, a solution is viewed as a set of values that can be assigned to a set of variables. Such approaches take advantage of puzzle constraints: given values are fixed; rows, columns and mini-grids are permutations [12]. Some of this interest in constraint satisfaction approaches may stem from the relationship that Sudoku has with Latin Squares and hence with the widely studied *quasi-group completion problem (QCP)* [13]. A Latin Square is an $n \times n$ table of n values, such that no value is repeated in any row and column; through the relationship with Latin Squares, Sudoku has been shown to be NP-Complete for higher dimensions [1]. A Latin Square is also a case of a quasi-group, having clear structure and both easy and hard instances, and as such it has been described as "an ideal testbed" [13] for the application of constraint satisfaction algorithms. QCP has been linked to many real-world problems and its study has revealed properties of search algorithms. A variety of constraint satisfaction approaches have been employed for Sudoku, such as the application of bipartite matching and flow algorithms [5], the use of forward checking and limited discrepancy search algorithms [13], and the application of Conjunctive Normal Form (CNF) in order to produce different encodings of the problem. The latter has been attempted for minimal and extended encodings (*i.e.* with redundant constraints) [10], and an optimised encoding that relies on knowledge of the fixed givens to reduce the number of clauses [11], an approach reported as useful in the solution of larger Sudoku grids.

An Integer Programming approach (the application of Linear Programming to a problem in which all constrained variables are integers) has been used in [6], which considered both puzzle creation and puzzle solution. A branch and bound algorithm was used to determine optimal solutions.

Lastly, an evolutionary approach to the solution of Sudoku puzzles has also been taken through the use of genetic algorithms [12]. In this, the authors designed geometric crossovers that are strongly related to the problem constraints. Much success was achieved with this approach, although genetic algorithms are generally not an ideal approach for the solution of Sudoku, because they do not directly exploit the problems constraints to reduce the examination of the full search space, as the authors acknowledge.

3 The Nature of the Problem

The Sudoku puzzle is deceptively complex. Although not instantly obvious, an empty Sudoku puzzle has an extremely large number of goal state solutions: 3,546,146,300,288 in total [14]. Even when givens are in place there is still an extremely large search space.

The approach to Sudoku grid solution taken in this paper is to employ optimization-based search, exploring sections of a search space that represents the possible moves from some initial state of the puzzle to a goal state, *i.e.* a solution to

that puzzle. All completed (solved) Sudoku grids are permutations of the numbers 1 to 9 in rows, columns and mini-grids, arranged so as to meet the puzzle constraints. The fixed givens in the initial state of the puzzle already meet the problem constraints. It is known *a priori* which values are missing from the initial grid (*i.e.* how many of each digit), and so these numbers may be placed in the missing spaces, in any order, to form an initial candidate plan, or 'incorrect solution'. Hence, a state in the search space is a structured collection of the correct puzzle objects, placed (with the exception of the givens) in the wrong order. From each state, a change to the state (or move) can be made by swapping any pair of non-given values. If there are *n* missing values from the initial Sudoku grid, there would be a maximum of *n(n-1)/2* permutations of those missing values, or moves possible from any state. This indicates how rapidly the state space increases in size with moves from an initial state.

The number of moves possible at each move, and hence the size of the search space, can be reduced however, without affecting whether a goal state can be located. By choosing to place in each mini-grid only those digits which are missing from that mini-grid, one of the problem constraints – that of ensuring that each mini-grid contains all 9 digits – will already be satisfied. In this case, the number of distinct states will be reduced to:

$$\sum_{i=1}^{n_r} \sum_{j=1}^{n_c} \frac{n_{ij}\,(n_{ij} - 1)}{2}$$

where n_{ij} is the number of non-given values in the mini-grid at row i, column j, n_r is the number of rows and n_c is the number of columns in the grid. This represents a reduction of:

$$\sum_{i=1}^{n_r} \sum_{j=1}^{n_c} \frac{n_{ij}\left(\sum_{a=1}^{n_r} \sum_{b=1}^{n_c} n_{ab} - n_{ij}\right)}{2}$$

in the number of moves possible from each state.

Further, the number of total possible combinations of non-given values, and therefore the number of distinct states in the search space (another indicator of search space size) is now reduced to the number of permutations of non-given values number within their respective mini-grids, *i.e.*

$$\prod_{i=1}^{n_r} \prod_{j=1}^{n_c} n_{ij}!$$

An example Sudoku grid, with 28 givens, is shown in Figure 1. This grid contains 6 mini-grids with 3 givens (and therefore 6 non-given values), 1 mini-grid with 2 givens (and 7 non-given values) and 2 mini-grids with 4 givens (and 5 non-given values). In total, 53 values must be added to form an initial state. Hence, under the scheme proposed above, from each state, 6×6(6-1)/2 + 7(7-1)/2 + 2×5(5-1)/2 = 131 moves (or swaps of pairs of non-given values) are possible – a reduction from 53×(53-1)/2 = 1378. A potential number of 53! combinations, or potential distinct states, has been reduced to 6×6! × 7! × 2×5! distinct states. This reduces the search space enormously, speeding any process of search.

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

Figure 1: An Example Sudoku grid

4 Modelling the Problem

The Sudoku puzzle is formulated here as a straightforward heuristic state-based search optimization problem, requiring definitions for an optimization technique, state representation, neighbourhood (chosen to reduce the search space), operators, and an objective function that employs information concerning the problem domain.

4.1 Search Technique

The search technique chosen was a modified steepest ascent hill-climbing algorithm, which employs an objective function to descend the search space making locally optimal decisions on which move to make next, always choosing the highest scoring successor in the entire neighbourhood of the current state [8]. Initially, a best-first search approach was employed, using a priority queue of as-yet unexplored states ordered by objective function score so that the choice of next move was not restricted to the neighbourhood of the current state, but may return to a previously located more promising state and thereby avoid local optima. The highest scored state in such a queue is the state to be expanded next. However, it was found that the queue was not necessary at all in the solution of the easier SuDoku puzzles, as descent is continuous. In the solution of more difficult puzzles, although the queue was occasionally used to backtrack in the search space, the method became stuck in large plateaus. The queue then became more a means of detecting plateaus than a method contributing to search, and was kept for that purpose (as described further in Section 4.5 below).

The basic algorithm is as follows [8]:

Current-state becomes initial state.
While goal state not found, or complete iteration fails to change Current-state, do:

1. *Let Best-successor be a state.*
2. *For each operator that applies to current state:*
   a. *Apply operator to determine a successor to current state*
   b. *Evaluate successor. If it matches goal state, return with success, else if it is better than Best-successor, change Best-successor to new state.*
3. *If Best-successor is better than Current-state, then Current-state becomes Best-successor.*

A test set of 100 different Sudoku puzzles, which varied in both level of difficulty and number of givens, was constructed. This test set included some puzzles with very hard ratings and a puzzle claimed to be the hardest Sudoku (called AI-Escargot and developed by Arto Inkala [15]).

The need for modifying this method to avoid local maxima and plateaus, and the approach taken to modification, are described in Section 4.5 (below).

4.2 State Representation and Operators

At each stage of the optimization, a current state is transformed by the single application of an operator to a selected successor, or neighbour, in the search space. All successors, or candidate solutions, are evaluated to determine the state with the highest evaluation.

As described in Section 3 (above), the operators are restricted to swaps of pairs of values within a mini-grid (excluding given values from any swap). Hence the neighbourhood is restricted to candidate plans differing only to the positions of two non-given values in a mini-grid. This avoids potential swaps of values between mini-grids that would worsen the solution by breaking one of the puzzle constraints, *i.e.* ensuring that each mini-grid contains all values between 1 and 9.

4.3 The Objective Function

An objective function, f, is required as a measure of the closeness of a given state to a goal state. The function is a weighted sum of n problem domain related factors f_i employing n corresponding weights w_i, the general expression of which is:

$$f = \sum_{i=1}^{n} w_i f_i$$

The factors, or sub-functions, f_i must be chosen as measurable quantities of closeness to goal, such that the function f increases in value the closer a state is to a goal state. These factors embody heuristics for improving states, based on observations on the problem domain.

For the Sudoku puzzle, it is proposed here that the factor f_1 is a count of the number of different digits that appear in each row and column of a current state of the Sudoku grid. States very close to a goal state will typically have few repetitions (or 'clashes') of digits in rows and columns, hence f_1 will score highly. In full, f_1 is defined as:

$$f_1 = \sum_{i=1}^{n_r} \left| \{ x : x = a_{ij}, j = 1..n_c \} \right| + \sum_{j=1}^{n_c} \left| \{ x : x = a_{ij}, i = 1..n_r \} \right|$$

where n_c is the number of columns, n_r is the number of rows, and a_{ij} is the value in the grid at row i, column j.

When considering the rows and columns of the Sudoku grid, in a given state that is not a goal state, there will be clashes of certain digits in some of those rows and columns. The factor f_1 provides a measure, inversely, of the *overall* number of such clashes. However, clashes involving givens (the pre-defined and fixed values which may not be moved) should be considered to be of greater importance than clashes between two non-given values, as it is known that the position of the non-given value must be incorrect. Therefore, a second factor, or sub-function, f_2 is employed to embody a clearer measure of this consideration. This factor is defined as a count of the number of clashes between any given and non-given digits, and so the factor f_2 will tend to reduce as the grid improves. In full, f_2 is defined as:

$$f_2 = \sum_{g_{pq} \in G} \left| \{ i : a_{iq} = g_{pq}, i = 1..n_r, i \neq p \} \cup \{ j : a_{pj} = g_{pq}, j = 1..n_c, j \neq q \} \right|$$

where G is set of all givens, g_{pq} at row p, column q.

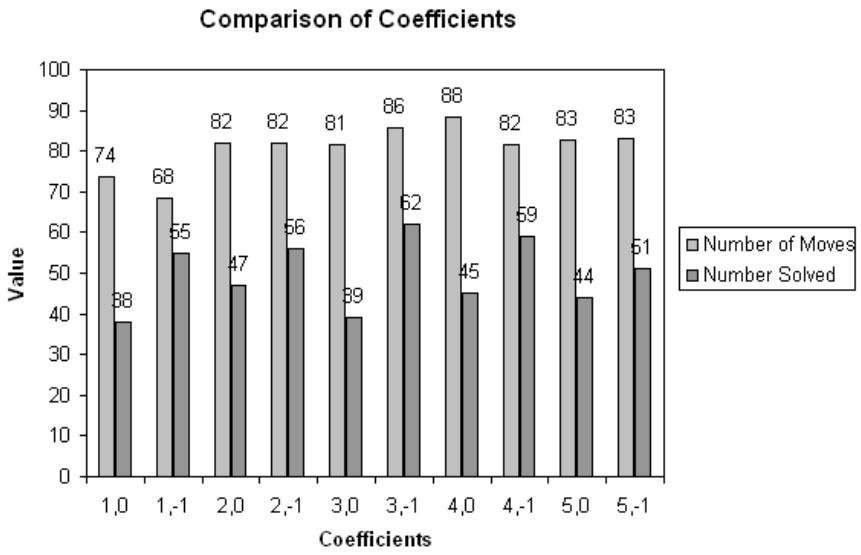

Figure 2 Performance of coefficient pairings

In this implementation of search, trial and error was used to determine appropriate values of the weights w_1 and w_2. The combination of values that led to solutions of a range of puzzles in the least number of iterations (or moves) was selected. (The number of iterations was chosen as it has a direct link to both the number of total successor states considered (as the number of successors in each state's neighbourhood is constant) and the overall time taken to reach a goal state.) A range

of coefficient values were tried for the solution of the test set of 100 Sudoku puzzles. For some coefficient choices, not all puzzles were solvable using this optimization approach, as search became mired in local optima and plateaus. The most successful coefficients were chosen on the basis of number of puzzles solved (*i.e.* goal state located), number of iterations, with results being shown in Figure 2 for a selection of the tested coefficient pairings. The pairing w_1=1 and w_2=–1 was chosen as, on average, it returned the goal state in the least number of iterations, and also solved a large number of puzzles.

4.4 The Initial State

Section 3 defined a state as being a structured collection of the right objects (*i.e.* the missing digits) placed in the empty spaces, such that each mini-grid contains all the digits 1 to 9 (albeit not necessarily in the correct order). These missing digits could be placed in any order so as to construct an initial state for the puzzle. However, a method that provides a good ordering that reduces the amount of search space to be examined, while having itself minimal processing cost, would be of great benefit. It is proposed here that a greedy algorithm approach is employed in the initial placement of the missing digits. A greedy algorithm makes locally optimum choice at each stage in the solution of a problem, in the hope that a global optimum may be reached [16].

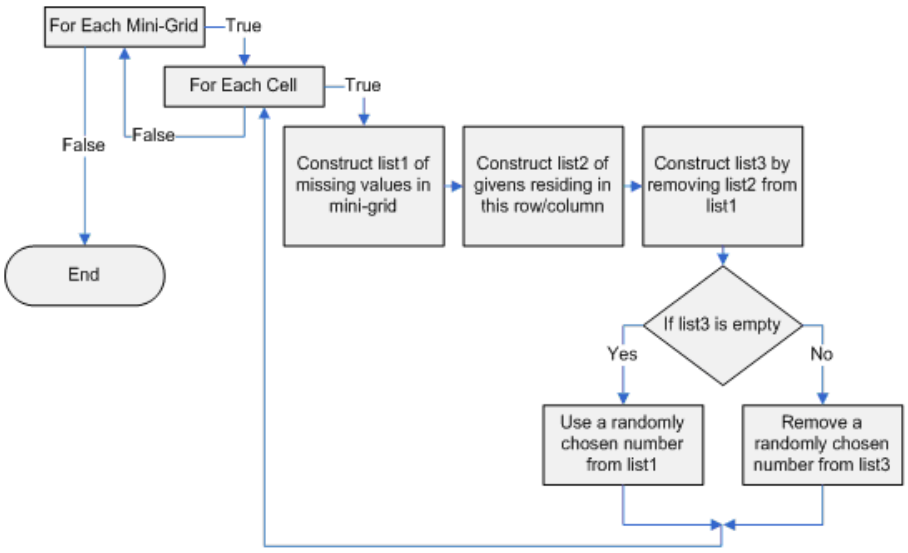

Figure 3 Greedy algorithm for initial placement of missing values

An initial state arrived at through random insertion of the missing digits is likely to contain large numbers of clashes. As a result, any solution to the puzzle is likely to require a large number of swaps. The greedy algorithm proposed here is intended to greatly reduce the number of clashes in the initial state, but not to fully solve the

puzzle. The algorithm is detailed in Figure 3. It places the missing numbers in each mini-grid in turn, filling each empty cell with the next available number that produced no clashes with the givens. At some stage within this process, it would be necessary to make placements that do incur clashes, but the overall number of initial clashes is greatly reduced. This employs a greedy algorithm that makes locally optimum choices at each placement, albeit with little realistic hope of finding a global optimum (as the algorithm is not then pursued to make further improvements to the overall placement).

In order to improve this initial placement further, a limited *post-swapping* was performed. All values which clash with a given are swapped with another value in that mini-grid that does not produce a new such clash (selecting other non-given values from the mini-grid, ordered randomly, until a suitable one is found or no more are available, in which case the clash remains). The improvement in performance due to this amendment, measured using those puzzles from the test set that were ultimately solved, is illustrated in Figure 4. This shows averaged measures of the algorithm performance (shown as number of iterations, or moves required to reach a goal), the improvement in the f_1 part of the objective function for the initial state, and the reduction in the number of clashes with givens in the initial state. The cost of performing the initialization is measured in milliseconds. (All tests were performed on a Viglen Intel Pentium 4, 3.40GHz computer, with 1.99GB RAM, using implementations in Java using JDeveloper 10g.) For the low cost in processing time, the greedy algorithm and post-swapping are considered worth performing. It is also worth noting that this improved initialization solved a small number of the easiest puzzles (that have many givens) without the need for subsequent search.

	Greedy algorithm alone	Greedy algorithm and post-swapping
Time to Initialise (ms)	2	5
Number Of Iterations	71	66
f_1 score	164	173
Remaining Clashes	59	26

Figure 4 Difference in performance due to greedy algorithm amendment

4.5 Modification to the Search Algorithm

The success of the hill climbing approach described in this paper is quite strongly related to standard measures of puzzle complexity. For simple Sudoku puzzles, the algorithm is capable of reaching a goal state in relatively few moves. For more difficult puzzles, the approach becomes trapped in local maxima and plateaus, as is common will hill climbing. This leads to an observation concerning the objective function constructed: although the heuristics do ensure, generally, a direct descent through the search space towards a goal, they actually measure relatively little concerning the problem domain. Hence, many states in the space can map to the same objective function value, and for more difficult Sudoku puzzles, it seems common to reach an intermediate state for which many states in its neighbourhood share the same highest objective function score. That is, a plateau is reached, from

which the algorithm cannot escape. In initial experimentation, a priority queue was employed to enable backtracking to earlier, more seemingly promising paths. However, this met with limited success, as the same high objective function score might be found in many different parts of the space. Instead, a different approach was taken.

Here, a random-restart hill climbing approach was implemented. The priority queue was used as a means of detecting plateau, such that once the first 10, highest scoring solutions all shared the same score, and none of those solutions had successors with better scores, it was assumed that a plateau had been reached. (Experiments with different cutoffs revealed that 10 was a sufficient number.) At such a point, the queue was cleared and a new initial state was generated (as described in Section 4.4 above). Search recommenced with the new initial state.

All Sudoku puzzles in the test set of 100 puzzles were solved successfully using this modified approach.

5 Evaluation and Conclusions

All tests were performed on a Viglen Intel Pentium 4, 3.40GHz computer, with 1.99GB RAM, using implementations in Java using JDeveloper 10g.

Runs of the algorithm were analyzed as to the time taken (in milliseconds), the number of iterations (moves) which had to be performed to complete each puzzle, and the number of random restarts required. Generally, the number of iterations increased with the rated complexity of the puzzle, as shown in Figure 5 (which shows averaged values for the number of givens, number of iterations, number of restarts and time to solve, for all 100 puzzles in the test set). The hardest of the puzzles required, typically, very large numbers of iterations. In all categories, a small number of puzzles required a number of restarts greatly uncharacteristic for that grouping, with all other puzzles completing in similar numbers of iterations.

Level	Average Number of Givens	Average Number of Iterations	Average Number of Restarts	Average Time to Solve (ms)
Easy	31.12	233.96	0.12	1443996.56
Medium	25.84	392.08	1.08	1256019.20
Hard	23.44	724.60	2.80	3110608.32
Hardest	20.33	3587.58	3.39	34921304.31

Figure 5 Results by Problem Complexity Rating

Similarly, the number of iterations generally decreased with the number of givens, with a small number of exceptions. A small number of puzzles in the category of most givens had already been solved by the initialization (Section 4.4 above).

Number of Givens	Average Number of Iterations	Average Number of Restarts	Average Time to Solve (ms)
16-23	1571.652	3.086957	12419636
24-30	548.3182	1.386364	2357723
30-36	64.56522	0.130435	46220.17
37+	1	1.333333	10.33333

Figure 6 Results by Number of Givens

While the objective function was simple, it proved a sufficiently effective measure of closeness to goal state, as demonstrated by the successful solution of all puzzles, and the lack of need for restarts in many of the simpler cases. However, for Sudoku puzzles of greater complexity, the objective function leads to search spaces that have wide plateaus in many regions, preventing the algorithm from reaching a solution from every initialization. Modification of the initialization had a great impact on the ability of the algorithm to reach a goal state, and for all puzzles tested, an appropriate initialization was found from which a direct path to a goal existed without being 'interrupted' by a plateau. (Indeed, initialization was a more important determinant in the speed with which solutions were found than experimentation with the objective function.)

The nature of the Sudoku puzzle would indicate that many meta-heuristic approaches (such as genetic algorithms, particle search or ant colony optimization [16]) that employ pools of solutions, and possibly means of mutating solutions to avoid local maxima, might be appropriate. However, this paper demonstrates that such elaborate schemes, with their tendency towards inefficiency, are probably not justified. The employment of heuristics and problem initialization that directly exploit features of the problem domain are sufficient for the reliable solution of puzzles, regardless of rated complexity and number of givens. Further, it is expected that approach to the construction of heuristics and problem initialization can be generalised to Sudoku grids of greater size.

References

1. Yato T and Seta T. Complexity and Completeness of Finding another Solution and its Application to Puzzles. In: Proceedings of the National Meeting of the Information Processing Society of Japan, IPSJ, Japan, 2002 (SIG Notes IPSJ-2002-AL-87-2)
2. Pegg E. Enumerating Sudoku Variations, available at http://www.maa.org/editorial/mathgames/mathgames_09_05_05.html, 2005
3. Dotu I., del Val A and Cebrian M. Redundant modeling for the quasigroup completion problem. In: Rossi, F. (ed.), Principles and Practice of Constraint Programming (CP 2003), Springer-Verlag, Berlin, 2003, pp 288-302 (Volume 2833 of Lecture Notes in Computer Science)
4. Gomes C and Shmoys D. The Promise of LP to Boost CP Techniques for Combinatorial Problems. In: Jussien N and Laburthe F (eds.), Proceedings of the

Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, CPAIOR, France, 2002, pp 291–305
5. Simonis H. Sudoku as a constraint problem. In: Hnich B, Prosser P and Smith B (eds.) Modelling and Reformulating Constraint Satisfaction Problems, Proceedings of the Fourth International Workshop, CP, 2005, pp 13-27
6. Bartlett AC and Langville AN. An Integer Programming Model for the Sudoku Problem. Preprint, available at http://www.cofc.edu/~langvillea/Sudoku/sudoku2.pdf, 2006
7. Gordon R. Minimum Sudoku. Internal Report. University of Western Australia, 2006
8. Rich E and Knight K. Artificial Intelligence (2nd Edition), McGraw-Hill: Singapore, 1991
9. Jones SK. Solving methods and enumeration of Sudoku. Final Year Project. University of Glamorgan, 2006
10. Lynce, I and Ouaknine, J. Sudoku as a SAT problem. In: Golumbic M, Hoffman F and Zilberstein S (eds.), Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics , AIMATH, 2006
11. Kwon G and Jain H. Optimized CNF Encoding for Sudoku Puzzles. In: Hermann M (ed.) Proceedings of the 13th International Conference on Logic Programming for Artificial Intelligence and Reasoning, available at http://www.lix.polytechnique.fr/~hermann/LPAR2006/short/submission_153.pdf , 2006
12. Moraglio A, Togelius J and Lucas S. Product Geometric Crossover for the Sudoku puzzle. In: Yen GG, Wang L, Bonissone P and Lucas SM (eds.), Proceedings of the IEEE Congress on Evolutionary Computation, IEEE Press, pages 470-476, 2006
13. Cazenave T and Labo IA. A Search Based Sudoku solver, available at http://www.ai.univ-paris8.fr/~cazenave/sudoku.pdf, 2006
14. Felgenhauer B and Jarvis F. Enumerating Possible Sudoku Grids. Internal Report. University of Sheffield, 2005
15. Inkala A. AI Escargot - The Most Difficult Sudoku Puzzle, Lulu Publishing, 2007
16. Michalewicz Z and Fogel DB. How to Solve It: Modern Heuristics, Springer: Berlin, 2000