Automation of the Solution of Kakuro Puzzles

R. P. Davies, P. A. Roach, S. Perkins

Department of Computing and Mathematical Sciences, University of Glamorgan, Pontypridd, CF37 1DL, United Kingdom, rpdavies@glam.ac.uk

Abstract

Kakuro puzzles, also called cross sum puzzles, are grids containing clues to the completion of numerical 'words'. Being structured in a similar way to crossword puzzles, Kakuro grids contain overlapping continuous runs that are exclusively either horizontal or vertical. The 'clues' take the form of specified run totals, and a puzzle is solved by placing a value in each cell such that every run sums to its specified total, and no run contains duplicate values. While most puzzles have only a single solution, longer runs may be satisfied using many arrangements of values, leading to the puzzle having a deceptively large search space. The associated, popular Sudoku puzzle has been linked with important real-world applications including timetabling and conflict free wavelength routing, and more recently, coding theory due to its potential usefulness in the construction of erasure correction codes. It is possible that Kakuro puzzles will have similar applications, particularly in the construction of codes, where run totals may form a generalised type of parity check. A project has begun to investigate the properties of the Kakuro puzzles, and thereby establish its potential usefulness to real-world applications including coding theory. This paper reports some early findings from that project, specifically concerning puzzle complexity and the appropriateness of heuristic approaches for its automated solution. It highlights the use of heuristics to guide search by a backtracking solver, in preference to local search optimisation, and reports on the effectiveness of two heuristics and a pruning technique for reducing solution time. The authors believe this to be the first published work in the use of heuristics, in combination with pruning, for the automated solution of Kakuro puzzles.

1 Introduction

Kakuro puzzles are number puzzles that have strong similarities with the more familiar crossword puzzle, due to their use of 'clues' to specify correct numerical 'words' within a grid structure. Unlike crosswords, Kakuro puzzles more easily transcend language barriers due to their use of number sequences. Puzzles of this

type typically consist of an $n \times m$ grid containing black and white cells. All white cells are initially empty and are organised into overlapping continuous runs that are exclusively either horizontal or vertical. A run-total, given in a black 'clue' cell, is associated with each and every run. The puzzle is solved by entering values (typically in the range $1,...,9$ inclusive) into the white cells such that each run sums to the specified run-total and such that no value is repeated in any horizontal or vertical run.

Most published puzzles consist of an $n \times m$ grid and are *well-formed* [6], meaning that only one unique solution exists. Such puzzles are also called promise-problems (the promise being a unique solution) [1]. The puzzles are designed so that this unique solution may be determined through the employment of a range of types of logical deduction and reasoning; no guesswork should be needed. Many puzzles have reflective or rotational symmetry, although this is only to improve the visual appearance of the grid.

The name 'Kakuro' comes from the Japanese pronunciation of the English word 'cross' appended to the Japanese word for 'addition'. The name was part of a re-branding by Japan's Nikoli Puzzles Group of Dell Magazines' 'Cross Sum' puzzles, which can be traced back as early as 1966 [4]. Currently, the popularity of Kakuro in Japan is reported second only to Sudoku puzzles [4], but it is only during the last four years that the puzzle has gained wider global popularity, particularly in the West.

Related puzzles include: 'Cryptic Kakuro' [11], in which alphametric clues must be solved as a prerequisite to the Kakuro puzzle itself; 'Cross-sum Sudoku' [11], which combines the rules of standard Kakuro puzzles with the constraints of standard Sudoku puzzles; 'Cross Products', in which 'clue' cells suggest the product of digits in a run, rather than their sum; and 'Survo Puzzles' [7], in which the values $1,...,mn$ must be placed, once each, into an $n \times m$ grid that often contains givens, so as to satisfy row and column sums.

The associated, popular Sudoku puzzle has been linked with important real-world applications including timetabling [5] and conflict free wavelength routing [3], and more recently, coding theory due to its potential usefulness in the construction of erasure correction codes [10]. At present, very little has been published specifically on Kakuro and its related puzzles. The authors have previously reported on the use of binary integer programming and local search approaches to the solution of Kakuro [2], concluding on the need for heuristics to guide solution and for the need to reduce the size of the search space to be examined. The solution of Kakuro puzzles has been shown to be NP-Complete [9], through demonstrating the relationship between Kakuro and the Hamiltonian Path Problem, and 3SAT (the restriction of the Boolean satisfiability problem). It is possible that Kakuro-type puzzles will have similar applications to Sudoku, particularly in the construction of codes, where run totals may form a generalised type of parity check.

A project has begun to investigate the properties of the class of Kakuro puzzles, and thereby establish its potential usefulness within a range of applications, in-

cluding coding theory. This paper reports some early findings from that project, specifically concerning puzzle complexity and the appropriateness of heuristic approaches for its automated solution. It highlights the use of heuristics to guide search by a backtracking solver, in preference to local search optimisation. Evaluation is presented of the effectiveness of two heuristics for guiding search and a method for pruning the search space that need be considered by the solver. The authors believe this to be the first published work in the use of heuristics, in combination with pruning, for the automated solution of Kakuro puzzles.

2 Problem Analysis

Let a Kakuro grid be termed K, where K has dimension $n \times m$, and the cell at row *i* and column *j* is termed $k_{i,j}$. Each cell is either a white cell (to be assigned a numerical value in the range $1,...,9$) or a black 'clue' cell. Grid K contains a collection of runs of white cells. Each of these individual runs is exclusively either horizontal or vertical and is termed a tuple r_l ($l = 1, ..., p$) where $r_l \in r$, the set of all tuples, and *p* is the number of runs contained in the puzzle grid. We define the tuple r_l to be such that it contains no repeated elements.

Therefore r_l is described either as a tuple of connected horizontal white cells or of connected vertical white cells. A horizontal run is defined:

$$r_l = (k_{i,j_s},..., k_{i,j_e}) \qquad k_{i,j_x} \in r_l,\ s \le x \le e$$

where the run is in row *i* ($1 \le i \le n$), beginning in column j_s and ending in column j_e, ($1 \le j_s < j_e \le m$). A vertical run is defined:

$$r_l = (k_{i_s,j},..., k_{i_e,j}) \qquad k_{i_x,j} \in \mathrm{r}_l,\ s \le x \le e$$

where the run is in column *j* ($1 \le j \le m$), beginning in row i_s and ending in row i_e ($1 \le i_s < i_e \le n$).

The values to be placed within each white cell, $k_{i,j}$, are governed by puzzle constraints, namely that:

- In each and every run, $r_l \in r$, the same value must appear in no more than one cell:

$$k_{i,j_u} \ne k_{i,j_v} \qquad k_{i,j_x} \in r_l,\ j_u \ne j_v$$

$$k_{i_u,j} \neq k_{i_v,j} \qquad k_{i_x,j} \in \mathrm{r}_l \ , \ i_u \neq i_v \ .$$

- In each and every run, $r_l \in r$, the corresponding run-total, t_l, must be satisfied:

$$\sum_{k_{i,j} \in r_l} k_{i,j} = t_l$$

Kakuro grids can vary in the difficulty of their solution. Generally, the complexity of a given puzzle cannot be determined by the size of the grid alone, but should instead be determined from a combination of the number of empty (white) cells in its initial state, the grid size and the magnitudes of the run-totals.

In order to establish the potential size of the search space for a puzzle, we could consider the number of options for assigning values to a cell and thereby determine an upper bound for the number of different grid arrangements of values. (We note that, for a well-formed puzzle, all but one of these grids, the unique solution, would be invalid due to the puzzle constraints.)

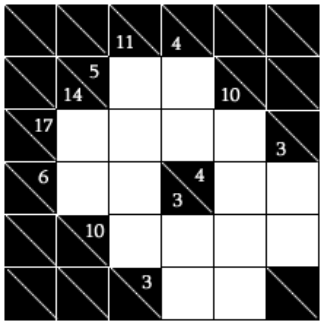

Fig. 1. A 5×5 initial puzzle grid

A crude upper bound for the number of grid arrangements for a puzzle grid with *w* white squares is 9^w, since each square can take any of nine numerical values, assuming the standard range of values $(1,...,9)$ is being used. The puzzle grid in Fig. 1, with sixteen white squares, would therefore have an upper bound of $9^{16} \approx 1.853 \times 10^{15}$ possible arrangements.

This upper bound is greatly reduced by considering which of the nine available values can legitimately be placed in each of the white cells, c_i, depending on the run(s), r_l (each with corresponding run-total t_l), in which the cell resides.

A set of values, P_l, that may be assigned to cells in a run r_l is constructed, such that:

$P_l = \{1, \ldots, 9\}$ if $t_l > 9$

$P_l = \{1, \ldots, a\text{-}1\}$ if $t_l = a$, $a \leq 9$

The improved upper bound would then be:

$$\prod_{i=1}^{w} \min\{|P_l| \mid c_i \in r_l\}$$

For example, the white cell at the uppermost left of the grid in Fig. 1 is a member of a run totalling 11 and of another totalling 5. Concentrating on the lower of the two run-totals, only a value in the range $1,...,4$ can be placed in this cell. When all cells are considered in this way, a new upper bound equalling 1,133,740,800 arrangements can be calculated for this example.

The positioning of runs, and the selections of run totals of Kakuro puzzles can vary greatly. This makes the task of devising a general formula for the exact number of possible Kakuro grid arrangements of a given size difficult, if not impossible, to achieve. Instead we focus on determining the total number of arrangements of values within a single run which would satisfy the puzzle constraints – the run total constraint, and the requirement to have no duplicated values in the run. Different sets of distinct values might meet the run constraints, but each set can be permuted into different orderings – only one of which will match the puzzle solution. The total number of such arrangements of values can be derived from the coefficients obtained from a series expansion of the generating function:

$$F(x) = |r_l|! \prod_{i=1}^{9} (1 + x^i a)$$

The coefficient of $a^{|r_l|} x^{t_l}$ represents the number of ordered cell value compositions of length $|r_l|$ that have no repeated value and that sum to t_l. This generating function has been used to develop a look-up table that is employed in a heuristic in Sect. 3.3.2. It is worth noting that the function is generalisable to Kakuro puzzles that use larger sets of assigned values (*i.e.* beyond $1,...,9$).

3 Automating the Solution

Automated approaches to the solution of a given Kakuro puzzle can be placed into two categories. One category of approaches would use similar methods to those used by a human solver, where the constraints of the puzzle (run-totals and non-

duplication of values within runs) are considered in turn in some logical order until a valid solution is found. Alternatively, the secondary category would use search algorithms, possibly along with heuristics and objective functions for optimisation. These heuristics and objective functions would incorporate problem domain information.

3.1 Selecting a Suitable Approach

A given Kakuro puzzle could be solved exhaustively. That is, all possible values are tried in all cells, fully enumerating the search space in order to locate the solution. This approach is adequate for smaller grids or when a smaller range of numbers is to be used but very time consuming and inefficient for most puzzle grids where very large numbers of puzzle states would have to be checked.

The puzzle constraints, non-duplication of values within runs and the summation requirement of values to a specified run-total, make the puzzle seemingly appropriate for a constraint-based approach to a solution. Binary integer programming is one such approach, and such a formulation of the puzzle has previously been presented by the authors [2]. In that formulation, ten binary decision variables, $A_{i,j,k}$, are associated with every cell, where row *i* and column *j* specify the cell position, and *k* specifies an available value for assignment to the cell (with zero indicating a black square). Puzzle constraints and trivial constraints (such as there only being one value per cell and only values in the range $1,\ldots,9$ can be added to white cells) are expressed explicitly. The solution is indicated by the collection of binary decision variables that are set to 1, showing which value *k* should be assigned to the cell at row *i* and column *j*. The results for this approach, using XPress MP (a suite of optimisation packages) showed that this approach works well for small puzzles [2]. However, the large number of binary decision variables for larger grids may make this an inefficient general approach.

The difficulty of search space size might be overcome by the use of heuristics in a local search optimisation approach [8]. This approach would employ some objective function to determine an efficient path through the search space, beginning from some *initial state*. This initial state might be a particular arrangement of values within cells such that run duplication constraints are met but not run-total constraints. An *operator* would then change the values within one or more of the cells so that *successor states*, different arrangements of values within cells, are produced following each iteration of the search (as illustrated in Fig. 2).

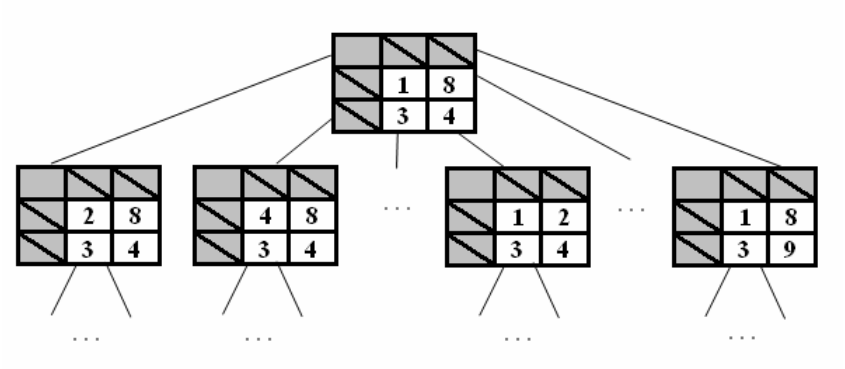

Fig. 2. A sample search space containing puzzle states

All states would then be evaluated and scored by the objective function, so that the state with highest 'score' would be explored next. However, such an approach is only feasible if a sensible and effective objective function can be constructed, such that it is possible to move reliably towards the goal state. Unfortunately, the amount of problem domain information that can usefully be incorporated into an effective objective function is limited. This puzzle information relates specifically to how closely the current horizontal and vertical run sums match the specified run-totals. There is a likelihood of many different states mapping to the same score and of the method becoming stuck in plateaus [8] in the search space. Similar difficulties have been reported in a local search optimisation approach to the solution to Sudoku puzzles [6]. Also, since each value in a particular cell can be replaced by up to eight alternative values, the search space can grow very rapidly. Solutions of larger puzzles would inevitably involve the storage of a very large number of states.

Meta-heuristic approaches might be employed to overcome the limitations of the objective function, however the authors wish to employ problem domain information more directly. For the above reasons, search optimisation approaches are not pursued here. Instead we employ a backtracking approach to solving Kakuro puzzles, as described below.

3.2 Backtracking Solver

An approach that takes more direct advantage of the problem complexity characteristics – notably the permutations of the values that may legitimately be assigned to runs – is desired. A backtracking algorithm, employing a depth-first approach to examining the search space, can be made appropriate for the solution of Kakuro puzzles if suitable heuristics to guide the backtracker, and effective pruning conditions can be determined to reduce search space size. In this section, a backtracking algorithm, implemented through the use of a stack, is described that incorporates

conditions to prune parts of the search space in which valid solutions will definitely not be located.

BACKTRACKING ALGORITHM

Initialise puzzle information and stack.
Current_State becomes the initial-state. Add Current_State to stack.
Current_Cell is set to be the first available white cell.
Current_Value = 1.
WHILE [empty white cells exist]
    Place Current_Value into Current_Cell.
    Increment Iteration_Count.
    Determine runs in which Current_Cell resides, and corresponding run totals.
    IF [no duplicates in runs] and ([run-total(s) not exceeded] or [run(s) completed correctly])
        Push Current_State to stack.
        IF [empty white cells exist]
            Current_Cell becomes next available cell.
        END-IF
        Reset Current_Value to 1.
    ELSE-IF ([runs under target run-totals] or [duplicate in run(s)]) and [Current_Value<9]
        Current_Value = Current_Value +1.
    ELSE
        Pop state from stack to become Current_State.
        Current_Cell becomes previous cell.
        Current_Value becomes value within Current_Cell.
        WHILE [Current_Value = 9]
            Pop state from stack to become Current_State.
            Current_Cell becomes previous cell.
            Current_Value becomes value within Current_Cell.
        END-WHILE
        Current_Value = Current_Value +1.
    END-IF
END-WHILE
Output Current_State as solution.

This approach begins with an empty grid and attempts assignments of values to each white cell in turn, starting with the lowest numerical value, and beginning the placements from the top leftmost cell. It follows a depth-first [8] enumeration of the search space, favouring the assignment of low numerical values, but tests within the algorithm ensure that some fruitless paths through the search space are avoided. An apparently successful assignment of a value to a cell (one which does not violate puzzle constraints) results in the current grid being pushed onto the stack. Violations of the puzzle constraints – a duplicate value in a run, or an ex-

ceeded or under-target run total where all possible values have been considered for the final cell of a run – result in the algorithm backtracking, and popping the last successful grid state from the stack. The stack only stores incomplete states, that are apparently valid, along one branch of the search space, thus avoiding the memory based issues which can arise in search approaches in which all valid partial states encountered are stored (for example in the queue of a local search optimisation approach [8]). An iteration count is incremented each time an attempt is made to assign a value to a cell, and is used as a measure of algorithm performance in Sect. 4.

While this approach is ideal for smaller puzzles, the algorithm can be required to perform a great deal of backtracking in larger puzzles. This necessitates the addition of further components. The heuristics and pruning conditions that have been tested in this project are described in Sect. 3.3 below.

3.3 Modifications to the Backtracking Algorithm

In this section, three modifications to the Backtracking Algorithm of Sect. 3.2 are proposed. The results of using these approaches are presented and analysed in Sect. 4.

3.3.1 Cell Ordering

It is proposed here that the path taken through the search space be guided by consideration of how many valid arrangements of values there are for each run. The *cell ordering* heuristic employed is that by favouring the completion of cells in runs having fewest valid arrangements, a reduction can be achieved in the maximum amount of backtracking required due to incorrect assignments to cells considered near the start of the search process. Those cells in runs having most potential valid arrangements will be considered later, tending to push the consideration of cells requiring most backtracking to a deeper level in the search space.

As an example, a run-total of 6 over two cells can be filled using the tuples (1, 5), (5, 1), (2, 4) and (4, 2). (The tuple (3, 3) would be invalid due to the duplication constraint). Hence this run can be filled in four different ways.

A *look-up* table is constructed using the generating function of Sect. 2. This table explicitly states how many distinct compositions of values exist for each run-total t_l and all possible run lengths $|r_l|$.

As this approach uses calculations based on entire runs, rather than single cells, a cell inherits the lowest number of choices of any run in which it is situated. This represents an upper bound for the actual number of choices for that cell. (We note that a more accurate measure is to be found in the intersection of the arrangements in runs, which is more difficult to calculate, and remains as future work.)

3.3.2 Reverse Value Ordering

This heuristic favours the assignment of values in the range $1,...,9$ in reverse order, essentially being based on the 'assumption' that puzzles will be solved more quickly in this manner. Clearly, all values are equally likely to be the content of a cell of a puzzle solution, in a general sense; the actual likelihood of, for example, a 1 or 9 appearing more frequently in a solution will be puzzle-specific. This is a poor heuristic, but no worse in general that the reverse assumption. Hence it provides a useful test of the performance of the algorithm, when measuring the results of many puzzles. A puzzle having several high values in cells considered at the start of solution will probably solve more quickly when using this heuristic.

3.3.3 Projected Run Pruning

The Backtracking Algorithm of Sect. 3.2 checks for invalid assignments to a run on the completion of that run. This will still allow poor choices of values to be placed at the beginning of a run, such that the run total can not be met with legitimate value assignments in the remaining cells. As an example, consider a run of 5 cells having the run total 35. A placement of 1 in the initial cell will seem legitimate, but even the assignment of the largest values to the remaining cells – 9, 8, 7 and 6 – will only lead to a total of 31. In such a case, considerable processing time would be wasted attempting to fill the remaining cells, until the Backtracking Algorithm eventually places a value larger than 4 in the initial cell. By considering whether a run can possibly be completed to meet its total, each time an assignment is made, fruitless branches of the search space can be pruned.

An additional validity check is added to the Backtracking Algorithm of Sect. 3.2. On assigning a value to a cell in a run that still possesses unassigned cells, a calculation is performed of the sum of the largest possible values that may still legitimately be added to the remaining cells of that run. If this sum would yield a run total at least matching the specified run total for that cell, the backtracker continues, otherwise this branch of the search space is pruned. This approach will reduce the number of puzzle states that need to be considered and hence should, in general, decrease the time taken to obtain a solution to a given puzzle.

4 Results and Timings

There is no published work with which to compare the findings of this project, and so the results obtained using the heuristics and the projected run pruning from Sect. 3 will be compared to results obtained using the backtracking solver alone, for specific puzzles of varying sizes. Tests were performed on a Viglen Intel Core

2 Duo processor 2.66GHz, with 2GB RAM. Programs were developed in Java (using Oracle Jdeveloper 10.1.3.3.0) and executed in the J2SE runtime environment.

Initial experimentation focused on establishing the relative and general effectiveness of the methods proposed in Sect. 3, and results are shown in Table 1. Few puzzles of small size were available for testing, but those tested were deemed sufficient to examine the methods and to demonstrate the puzzle-specific nature of their effectiveness. The numbers of iterations (explained in Sec. 3.2) are shown for a range of puzzle sizes.

Table 1. Iteration counts for specific puzzles, in each method

		Heuristic Used				
		Backtracking Alone	Cell Ordering	Value Ordering	Projected Run Pruning	Projected Run Pruning & Cell Ordering
Puzzle Grid Size	2×2	96	96	16	42	42
	3×3	68	22	60	60	22
	4×4	444	311	40	86	131
	5×5(a)	142	213	309	142	213
	5×5(b)	2,917	209	2,562	2,383	100
	5×5(c)	983	423	424	149	111
	5×5(d)	2,735	237	1,353	429	195
	6×6	210	650	675	210	650
	7×7	20,393,677	1,052,747	495,945	12,455,461	24,636
	8×8	14,347	71,168	3,140	7,032	27,440

As would be expected, the reverse value ordering worked best on certain puzzles – these being ones in which the first few cells that considered had high values. Cell ordering was often effective, but seemed less so for larger puzzles – for certain puzzles it performed worse than backtracking alone. In contrast, the projected run pruning performed more consistently, never requiring more iterations than the backtracker alone (as would be expected), and often requiring far fewer iterations. A method that encourages rapid and early pruning is desired. The combination of cell ordering and projected run pruning occasionally reduced the number iterations below the count achieved by either approach individually, suggesting that the methods might combine well in guiding the search method to earlier pruning of the search space. However, this behaviour was not consistent.

Puzzles of small size generally solve quite rapidly, but the processing overhead of the methods is of interest here. Table 2 shows the average time taken per iteration, measured in milliseconds, for the puzzle set of Table 1, this time banded accord-

ing to puzzle size. As would be expected, the average time per iteration is generally higher for smaller puzzles, as the search spaces are small, hence the benefits of pruning are less significant. It seems reasonably clear that the processing overheads of cell ordering (arising from a pre-processing step and indexing of an array) and pruning are small.

Table 2. Average time (milliseconds) taken per iteration

		Heuristic Used				
		Backtracking Alone	Cell Ordering	Value Ordering	Project Run Pruning	Projected Run Pruning & Cell Ordering
Puzzle Grid Size	< 5x5	3.5107	3.8425	3.2736	3.6641	3.6108
	5x5 & 6x6	3.3865	3.7496	3.4823	3.5374	3.6190
	> 6x6	3.4648	3.5826	3.4903	3.5685	3.5518

While few puzzles of small sizes are available, a larger number of published puzzles exist for a more 'standard' challenge. For a test set of puzzles of size 9×9, we pursue the most promising methods of projected run pruning and its combination with cell ordering. Table 3 shows results for thirteen puzzles of grid size 9×9.

Table 3. Iteration statistics for thirteen puzzles with grid size 9x9.

	Minimum Iterations	Maximum Iterations	Median Iterations	Average Iterations	% of cases where method performed best
Projected Run Pruning	5,829	1,554,208	65,760	256,455	76.92%
Projected Run Pruning & Cell Ordering	2,543	28,039,107	284,512	5,795,832	23.08%

In a small number of cases, the combination of the cell-ordering heuristic and pruning improved results (shown by the minimum number of iterations and the percentage of cases where improvement occurred), but the median and maximum number of iterations show both that the combination is an unreliable approach and that on certain puzzles, performance is greatly worsened. Hence the projected run pruning method is considered here to be the most reliable approach.

An extended test set of 20 puzzles of size 9×9 were solved using just projected run pruning. The fastest solution time (in milliseconds) was 21,096, the longest 36,075,603, the median 323,694 and the average 4,675,317. The average time per iteration was 3.6738 ms to 4 decimal places. This approach is relatively promising, but further pruning methods to force an earlier and more rapid reduction in search space size, and heuristics to guide search, are sought to enable more rapid solution.

5 Conclusion

This paper has analysed the size of a Kakuro search space. This includes establishing an improved upper bound for the number of possible arrangements of values in a Kakuro grid. More significantly, a generating function has been presented to determine the exact number of valid arrangements of values in any given run; this function can be used for different grid sizes and different numbers of values to be assigned. The suitability of a range of search approaches for the solution of Kakuro has been considered, and a backtracking approach has been presented as the preferred approach. A cell ordering heuristic, based on the number of valid arrangements of values in a given run, has been proposed and evaluated. Lastly, a pruning method has been proposed to reduce the part of the search space that need be examined, by checking whether a run total can possibly be met each time an assignment is made to a cell in that run.

The pruning method proved to be most effective in reducing solution time for a range of puzzle grids. The cell ordering heuristic performed unreliably, making reasonable improvements in the solution time in some cases, but greatly increased solution time in other cases. This heuristic might be improved by establishing the intersection of the arrangements in runs, rather than allowing a cell to inherit the lowest number of possible arrangements of valid solutions of the two runs in which it is situated, thus giving a better measure of possibilities for a single cell. The completion of a started run, in preference to continually jumping to the next single cell with fewest apparent choices, might also allow the earlier detection of fruitless branches.

The usefulness of Kakuro for applications, including Coding Theory, will depend in part on the development of methods to reliably enumerate the search spaces of specific puzzles more rapidly. A more detailed understanding of the size of the search spaces of puzzles will also be required. For this, it is proposed that the results of this paper be extended through further improvements to the upper bound for the number of possible arrangements of values in solution grids, through consideration of the intersection of runs.

Acknowledgments The authors wish to thank Sian K. Jones for many helpful discussions relating to this work.

References

1. Cadoli, M., Schaerf, M.: Partial solutions with unique completion. Lect. Notes Comput. Sci. **4155**, 101-10, (2006)
2. Davies, R.P., Roach P.A., Perkins, S.: Properties of, and Solutions to, Kakuro and related puzzles. In: Roach, P., Plassman, P. (eds.) Proceedings of the 3rd Research Student Workshop, University of Glamorgan, pp. 54-58 (2008)
3. Dotu, I., del Val, A., Cebrian, M.: Redundant modeling for the quasigroup completion problem. In: Rossi, F. (ed.), Principles and Practice of Constraint Programming, CP 2003 (Lect. Notes Comput. Sci**. 2833**), Springer-Verlag, Berlin, pp 288-302 (2003)
4. Galanti, G.: The History of Kakuro,. Conceptis Puzzles (2005). http://www.conceptispuzzles.com/articles/kakuro/history.htm. Cited 22 Feb 2008
5. Gomes, C., Shmoys, D.: The promise of LP to boost CP techniques for combinatorial problems. In: Jussien, N., 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, pp 291–305 (2002)
6. Jones, S.K., Roach P.A., Perkins S.: Construction of heuristics for a search-based approach to solving Sudoku. In: Bramer M., Coenen F., Petridis M. (eds) Research and Development in Intelligent Systems XXIV: Proceedings of AI-2007, the Twenty-seventh SGAI International Conference on Artificial Intelligence, pp. 37-49 (2007)
7. Mustonen, M.: On certain Cross Sum puzzles. Internal Report. (2006) http://www.survo.fi/papers/puzzles.pdf. Cited 22 Feb 2008
8 Rich, E., Knight, K.: Artificial Intelligence, 2nd Edition. McGraw-Hill, Singapore (1991)
9. Seta, T.: The complexities of puzzles, cross sum and their another solution problems (ASP),. Senior thesis. Dept. Information Science, University of Tokyo (2002)
10. Soedarmadji, E., McEliece, R.: Iterative decoding for Sudoku and Latin Square codes. In: Forty-Fifth Annual Allerton Conference, Allerton-07, University of Illinois (2007)
11. Yang, X.: Cryptic Kakuro and Cross Sums Sudoku. Exposure Publishing (2006)