The Use of Problem Domain Information in the Automated Solution of Kakuro Puzzles

Ryan P. Davies, Paul A. Roach, *Member IAENG*, and Stephanie Perkins *

Abstract—**Kakuro puzzle grids consist of overlapping continuous runs (collections of adjoined white cells) that are either exclusively horizontal or vertical. Constraints offer clues to their completion, achieved through the legal placement of values in the cells. To ascertain any potential applicability of Kakuro for real-world problems, it is first necessary to establish the underlying mathematical properties of the puzzle. Domain information is used to inform the development of algorithms for the automated solution of Kakuro puzzles through the application of search and pruning techniques, and inferences concerning problem properties are drawn. Results obtained are used to propose a method applicable to both smaller puzzles and larger counterparts, indicating the most effective automated approach.**

Keywords: Kakuro, Cross-Sum, logic puzzle, search, heuristics

1 Introduction

Kakuro puzzles, also called Cross-Sum puzzles, consist of an $n \times m$ grid containing black and white cells. The initially empty white cells are organised into overlapping continuous *runs* that are exclusively either horizontal or vertical. A *run-total*, usually given in a black "clue" cell, is associated with each run; the puzzle is solved by entering values (typically in the range $1, \ldots, 9$ inclusive) into the white cells such that each run sums to the specified run-total and no digit is duplicated in any run. Assuming only numbers in the range $1, \ldots, 9$ are used, a run can be between one and nine cells in length with a corresponding run-total in the range $1, \ldots, 45$; the majority of published puzzles contain runs that are at least two cells in length. Most published puzzles are *well formed* [1], meaning that a unique solution exists.

*Manuscript received November 17, 2009.

R. P. Davies is with the Combinatorics & Heuristics Group, Department of Computing and Mathematical Sciences, University of Glamorgan, Pontypridd, CF37 1DL, United Kingdom (email: rpdavies@glam.ac.uk).

P. A. Roach is with the Combinatorics & Heuristics Group, Department of Computing and Mathematical Sciences, University of Glamorgan, Pontypridd, CF37 1DL, United Kingdom (phone: (0)1443 482258; fax: (0)1443 482169; email: proach@glam.ac.uk).

S. Perkins is with the Combinatorics & Heuristics Group, Department of Computing and Mathematical Sciences, University of Glamorgan, Pontypridd, CF37 1DL, United Kingdom (email: sperkins@glam.ac.uk).

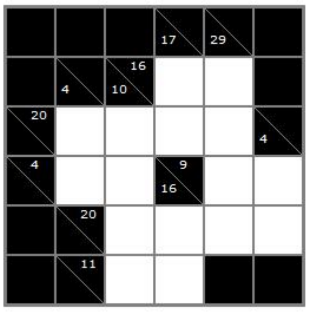

Figure 1: A typical Kakuro puzzle grid

The name "Kakuro" is derived from *"Kasan Karuso"*, the Japanese word for "addition" appended to the Japanese pronunciation of the English word "cross". This name was a part of a rebranding by Japan's Nikoli Puzzles Group of Dell Magazines' "Cross Sum" puzzles, as they were then known. Such puzzles were published as early as 1966 [2] by Dell Magazines. However, Kakuro's huge popularity is recent; Kakuro puzzles first appeared the United Kingdom on September 14th 2005 in The Guardian newspaper.

The popularity of Kakuro in Japan is now reported second only to Sudoku [2]. Sudoku puzzles have been linked with important real-world applications including timetabling [3], conflict free wavelength routing [4], experimental design [3] and more recently, coding theory due to its potential usefulness in the construction of erasure correction codes [5]. At present, very little has been published specifically on Kakuro, however, their solution has been shown to be NP-Complete [6] 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 may have similar applications to Sudoku. A project [7] has been conducted to establish the properties of Kakuro puzzles, since an understanding of the usefulness of these puzzles may arise from detailed analysis of their underlying properties. Both Kakuro and Sudoku contain some form of non-duplication constraints, however Kakuro puzzles additionally possess run-total constraints, meaning placed values not only have to be distinct from others within a run, but must sum to the correct target. Kakuro puzzles do not contain given values. Values do not necessarily appear a specified number of times within a puzzle grid and may appear more than once in a row or column; the repeated values must appear in distinct runs within that row or column.

As part of an investigation into the possible applicability of Kakuro to real-world problems, it was necessary to develop methods for the efficient automated solution of Kakuro puzzles. In particular, the authors considered whether puzzle domain information could usefully be employed to speed solution time. This paper describes the findings of that investigation.

Section 2 considers the size and complexity of the search space for an automated Kakuro puzzle solver. The knowledge gained is then employed in Section 3 to determine the most appropriate basic standard search technique to pursue. An implementation of that technique is outlined in Section 4, and improvements to it that are suggested by puzzle domain information are discussed. The results of the most promising approaches are compared in Section 5. Section 6 concludes on the effectiveness of using puzzle domain information to increase the efficiency of an automated solver, briefly outlining the consequences this may have for the potential application of Kakuro to the solution of real-world problems.

2 Problem Size and Complexity

Let a Kakuro grid be termed K, where K has dimension $n \times m$, and the cell at row i and column j ($1 \leq i \leq n$, $1 \leq j \leq m$) be termed $k_{i,j}$. Each cell is either a white cell (to be later assigned a numerical value from a given, valid range, usually $1, \ldots, 9$) or a black cell. White cells within grid K belong to the set W and collectively form runs, each of which are exclusively either horizontal or vertical. Each run is represented as a tuple r_l ($l = 1, \ldots, p$), where $r_l \in r$, the set of all tuples, and p is the number of runs contained in the puzzle grid. The tuple r_l is defined here to contain no repeated elements. Let $\mid r_l \mid$ be the number of cells contained in run r_l. Most black cells contain numerical "clues"; these are the run-totals ($t_l \in t$) that correspond to the runs within the puzzle grid that vertically or horizontally follow it.

A horizontal run is defined:

$$r_l = (k_{i,j_s}, \ldots, k_{i,j_e}) \qquad k_{i,j_x} \in r_l,\ 1 \leq j_s \leq j_x \leq j_e \leq m$$

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

$$r_l = (k_{i_s,j}, \ldots, k_{i_e,j}) \qquad k_{i_x,j} \in r_l,\ 1 \leq i_s \leq i_x \leq i_e \leq n$$

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

In order to gain an understanding of puzzle complexity, first consider how many ways there are to complete an individual run in a way which would satisfy the puzzle constraints - the run-total constraint, and the non-duplication requirement. Assuming the puzzle is well-formed, while different sets of distinct values might meet the run constraints, and each set can be permuted into different orderings, only one permutation of the set will match the puzzle solution.

Lemma 1. *The total number of valid, unordered arrangements of $\mid r_l \mid$ distinct values ($\mid r_l \mid \leq x$) for a specific run, r_l, with run-total t_l, is given by the coefficients of $a^{|r_l|}y^{t_l}$ obtained from the series expansion of the generating function:*

$$\prod_{i=1}^{x}(1 + ay^i)$$

Proof. The number of distinct, unordered arrangements of values for all run-lengths $\mid r_l \mid \leq x$, that have run-total t_l, is equivalent to the number of distinct integer partitions of t_l, given by the coefficients of y^{t_l} in the series expansion of the generating function [8]:

$$\prod_{i=1}^{x}(1 + y^i)$$

This generating function does not take into account the number of ways a run-total, t_l, can be partitioned into a specific number of distinct parts, or a specific run-length.

Since each block of the partition may be described using the dummy variable a, the number of occurrences of a (the power of a) describes the number of blocks in the partition and hence the number of cells present in the run. Therefore, the coefficients of $a^{|r_l|}y^{t_l}$ describes the number of partitions of t_l into $\mid r_l \mid$ parts or the number of arrangements of $\mid r_l \mid$ distinct values in a run of length $\mid r_l \mid$ with run-total t_l. □

Most published puzzles use the range of values $1, \ldots, 9$, *i.e.* $x = 9$.

Lemma 2. *Let q be the number of valid, unordered arrangements of $\mid r_l \mid$ distinct values ($\mid r_l \mid \leq x$) for a specific run r_l with run-total t_l, then the number of ordered arrangements of these values is given by:*

$$q \mid r_l \mid!$$

Proof. There are $\mid r_l \mid!$ ways of ordering $\mid r_l \mid$ distinct values. □

For example, the run $(k_{2,4}, k_{2,5})$ in the grid of Fig. 1 consists of two cells with a run-total 16. Following expansion of the generating function of Lemma 1, the coefficient of $a^2x^{16} = 1$; there is one unordered way of completing the run. However, the ordering of values within a run is important; multiplying this result by $\mid r_l \mid! = 2!$, there are two ordered ways of completing this run. Similarly, the run $(k_{5,3}, k_{5,4}, k_{5,5}, k_{5,6})$ in the grid of Fig. 1 consists of four cells with a run-total 20. Following expansion of the generating function of Lemma 1, the coefficient of $a^4x^{20} = 12$; there are twelve unordered way of completing the run but $12\times \mid r_l \mid! = 12 \times 4! = 288$ ordered ways of completing this run.

Instead of individual runs, full puzzles will now be looked at to establish bounds on the number of valid arrangements of digits from the range $1, \ldots, 9$ that may be placed into a grid. The search space, where *states* within such search space represent a partial assignment of values in a grid, can become very large, particularly for larger puzzle grids. Therefore, the puzzle constraints must be used to avoid arriving at value arrangements (states) that contain some arrangement, or partial arrangement of values that can definitely not be in the solution. States and search spaces are described in more detail in Section 3. Depending on the constraint(s) currently enforced, a given grid may possess a number of valid "solutions". If both puzzle constraints are relaxed, each white cell within the puzzle could potentially accept any of the nine values in the standard range $1, \ldots, 9$. A trivial upper bound, U_1, on the number of valid arrangements of values in a Kakuro puzzle grid is given:

Lemma 3. *Let w be the number of white cells within a Kakuro grid. Then the number of valid arrangements of values in a Kakuro grid is $U_1 \leq 9^w$.*

For the sample puzzle grid of Fig. 1, which possesses sixteen white cells, $U_1 \leq 9^{16} \approx 1.853 \times 10^{15}$ possible arrangements of values.

Kakuro puzzle grids vary greatly in terms of the run-totals that appear, however all grids adhere to the non-duplication constraint. If the run-total summation constraint is temporarily relaxed, a lower bound, L_2 on the number of arrangements of values in a Kakuro puzzle is derived:

Lemma 4. *The number of valid ways of placing values from the range $1, \ldots, 9$, into an $n \times m$ grid of white cells, such that no value is duplicated in any run is lower bounded by:*

$$L_2 \geq \prod_{x=9-n+1}^{9} \prod_{i=0}^{m-1} (x - i) \qquad (1)$$

Proof. L_2 depends on the lowest possible number of values that a cell can validly accept without contradicting the non-duplication constraint. Cells within the same horizontal or vertical run must contain distinct values. However, two cells placed diagonally with respect to one another, *i.e.* that are not within the same horizontal or vertical run, may accept the same value. If cells are considered from left to right and top to bottom, a cell at the intersection of a particular horizontal and vertical run may contain any value (from the standard range) that is not already present in the horizontal or vertical runs in which it resides. L_2 is derived by letting every adjacent cell to the cell in question, from both runs, contain a distinct value.

The first cell, $k_{1,1}$, in row 1, to be considered in a square grid may always contain any of the nine values from the standard range $1, \ldots, 9$. Cells below this initial cell, can therefore accept a consecutively lower value until the value $9 - n + 1$ is reached in the bottom cell. Consider each row in turn and let the first element in the row be x. Each cell in the row can then accept a consecutively lower value than that placed in the left adjacent cell until the cell in the rightmost column is reached, taking the value $(x - i)$ where $i = (m - 1)$. The lower bound, L_2, on the number of valid arrangements of values that exists is the product of the number of values each cell can accept. □

The above lower bound applies only to $n \times m$ grids of white cells. Actual Kakuro puzzle grids do not typically consist of an $n \times m$ grid of white cells; they may contain black cells at the corners or sides of the grid, or may contain internal black cells. The puzzle grid structure can therefore vary widely between puzzles and so is highly puzzle specific. However, similar upper and lower bound for an actual Kakuro puzzle grid, containing black cells, can be found. For the sample puzzle grid of Fig. 1, a lower bound similar to L_2 is $9^2 8^5 7^4 6^2 5^2 4^1 = 2.29 \times 10^{12}$ possible arrangements of values. The number of valid grids, where validity here ignores the run total constraints, was found by the use of an exhaustive counting program [7]. This algorithm attempts to assign all possible values to all possible white cells, providing there were no duplicate values within horizontal and vertical runs. Each time a "successful" assignment was made, a counter is incremented. Using this counting program, the grid of Fig. 1 has 32,917,207,692,096 such arrangements.

The current (trivial) upper bound, U_1 does not take into account any puzzle constraints while the current lower bound, L_2 does not take the run-total summation constraints into account; cells within a run must contain distinct values and values must meet required run-totals. Therefore, these are very high. When the run-total constraints are now taken into account, it is expected that upper bounds can be derived that are much lower, since the values that may actually be placed in cells are likely to be far more limited. In general, most cells cannot contain all values in the standard range $1, \ldots, 9$, particularly runs that sum to a very high or very low run-total. The bounds can therefore be greatly reduced by considering which of the nine available values can legitimately be placed in each of the white cells, depending on both runs in which the cell resides. (For example, if a run of length two had an associated run-total of six, the value 3 could not be placed in either cell belonging to this run because it would result in a duplicated value.) Each white cell within the puzzle grid belongs to two runs (one horizontal and one vertical), so a cell, $k_{i,j}$, can possibly contain values from two *candidate sets*: $C_{i,j,1}$, corresponding to the horizontal run and $C_{i,j,2}$, corresponding to the vertical run.

Lemma 5. *Let $C_{i,j,1}$ contain the values in the candidate set belonging to the horizontal run and $C_{i,j,2}$ the vertical run for cell $k_{i,j}$. The number of valid arrangements of values in a Kakuro grid is:*

$$U_2 \leq \prod_{\forall k_{i,j} \in W} \{\mid C_{i,j,1} \cap C_{i,j,2} \mid\} \qquad \textit{where } C_{i,j,1} \cap C_{i,j,2} \neq \emptyset$$

Proof. Let $C_{i,j,1}$ contain the values in the candidate set belonging to the horizontal run and $C_{i,j,2}$ the vertical run, such that $C_{i,j,1}, C_{i,j,2} \subseteq \{1, \ldots, 9\}$ for each cell $k_{i,j}$. Therefore, the actual possibilities for the values that may be placed into a particular cell, $k_{i,j}$, are from a set obtained through the intersection of these two candidate sets: $C_{i,j,1} \cap C_{i,j,2}$. U_2 then follows by taking the product of the sizes of all such intersecting candidate sets. □

In the grid of Fig. 1, the white cell $k_{3,2}$ is a member of a horizontal run (r_{l_1}) with run-total $t_{l_1} = 20$ and a vertical run (r_{l_2}) with run-total $t_{l_2} = 4$. A run-total of four over two cells can only be obtained by using an arrangement of $\{1, 3\}$, so the cell in question can take any of the values in the candidate set $C_{3,2,2} = \{1, 3\}$ based on run r_{l_2}. Similarly, a run-total of twenty corresponds to the candidate set $C_{3,2,1} = \{1, \ldots, 9\}$ based on run r_{l_1}. Therefore, since $\{1, 3\} \cap \{1, \ldots, 9\} = \{1, 3\}$, so only one of these two values can be added validly to this cell. The cell would therefore be assigned a "score" of two. When all cells are considered in this way, an improved upper bound $U_2 \leq 4^4 3^2 2^9 = 1,179,648$ arrangements can be calculated for this particular example.

This is an indication of the size of the search space for some automated solver. Hence, in any method of automated solution of puzzles, there is a clear need to reduce the size of search space that must be enumerated in order to locate the solution. The next Section uses this knowledge of likely search space size to establish as sensible choice of search technique, and Section 4 investigates how puzzle domain knowledge can be used to reduce the size of search space that must be enumerated.

3 Selecting a Search Approach

A Kakuro puzzle has a form suitable for a state representation. Any problem that can potentially be solved using a formal search-based approach must have [9]:

- A state representation, including an *initial state* (or initial position of the problem), a *goal state* (or states) and intermediary states;
- A goal test, that indicates when a solution has been reached;
- A set of operators that map one state to another so as to produce successor states.

A state may be represented by using an *incomplete state formulation*, in which a state is a partial solution to the problem that is to be completed, or by using a *complete state formulation*, in which a state is filled with a set of values that may be thought of as an 'incorrect' solution to be improved. Both formulations lead to a clearly defined *search space* [9], a tree of states or nodes, as the set of either all possible partially-filled or filled grid arrangements. The root of the tree is the initial state, and the goal, or goals, reside in one or more positions further down the tree. Each link between states represents one legal application of an operator, *i.e.* a move.

The most basic approaches to state-based search are exhaustive, or uninformed, search methods, such as *breadth-first* search and *depth-first* search [9, 10]. These are control strategies that dictate the order in which states are explored, *i.e.* added to the search space. Implementations of the depth-first search approach are generally efficient because only the states in the currently explored branch need to be stored. If it can be determined, though the use of problem domain knowledge, that a solution cannot lie further down a branch, that fruitless branch can be *pruned* - meaning that no further states along that branch are explored. This pruning is most commonly implemented through *backtracking*, *i.e.* returning to a parent node, which is usually the previously-explored state. For a Kakuro puzzle, the number of occurrences of each value is not known *a priori*, complicating the iterative alteration of a complete state formulation. An incomplete state formulation is more straightforward; potentially valid assignments to cells are made at each stage, the depth of the tree is equal to the number of cells within the puzzle and the solution must lie on the deepest level.

The goal test determines whether the puzzle has been solved, or whether it is necessary to backtrack. Therefore, an exhaustive control strategy may potentially fully enumerate the search space in order to locate the solution. Since Kakuro puzzles should be well-formed (and so possess a unique solution), the algorithm will generally terminate prior to full enumeration. However, full enumeration would be necessary to ensure uniqueness of solution. For grids having large numbers of cells, exhaustive methods will inevitably be inefficient and time consuming due to the number of states that would need to be considered. However, the placement of many values will lead to violations of the puzzle constraints, making domain information potentially useful for pruning techniques to speed solution.

In a *local search* approach, the local neighbourhood of a state is found by applying all valid operators to the current state to derive successor states. The merit, or score, of each successor is evaluated using an *objective function*, and the best successor state is selected for further expansion [10]. For Kakuro, the success of this approach depends on determining an objective function that can reliably move towards a goal state. Empty cells will complicate the scoring of the state; it would be unclear how a scoring mechanism should distinguish adequately between two states both having most cells empty, for example. An incomplete state formulation is therefore not ideally suited to a local search approach. By comparison, it seems more intuitive to distinguish between two completely filled grids, which may be thought of as representing 'poor' or incorrect solutions. However, the limited amount of information that can usefully be incorporated into an effective objective function is detrimental to the effectiveness of such a function. A local search approach would become trapped in local optima and in plateaus in the search space [10, 9], the latter arising from many adjacent successor states being assigned the same score. A similar difficulty has been reported for Sudoku [1].

Metaheuristics are used to solve various computational problems by adding a high-level algorithmic approach that guides existing control strategies and heuristics in a search for feasible solutions. Specifically, metaheuristic approaches might be employed to overcome the limitations of the objective function. Due to the numeric nature of Kakuro puzzles, a *genetic algorithm* [10] approach may be effective in automating their solution. Each puzzle state (chromosome) may be represented by a bit string where some mapping function exists to map each bit string to the familiar grid structure. The length of the bit string relates to the number of white cells within the grid, and is therefore constant. Crossover and mutation operations preserve desirable characteristics of 'fit' 'parent' genes to 'breed' new genes, while also injecting some new information through random mutations. However, as with an objective function in a local search techniques, a fitness function may not be effective since there is little puzzle domain information to utilize. Many chromosomes, representing solutions of differing "quality", would therefore receive the same fitness score. Metaheuristic approaches generally involve high processing overheads. Such elaborate and often inefficient schemes are probably not justified for puzzles in which a solution is both easy to define and relatively easy to locate within its search space.

Local search and metaheuristic approaches are not deemed suitable for Kakuro puzzles. Exhaustive approaches will always, eventually, locate a solution, and take direct advantage of the problem complexity characteristics of Kakuro puzzles, notably the permutations of the values that may legitimately be assigned to runs. However, to ensure efficiency of solution, particularly for larger or complex puzzles, the addition of effective pruning is required. It is expected that the use of pruning within a depth-first backtracker will lead to an efficient approach to the automated solution of Kakuro puzzles and will enable puzzle properties to be highlighted. This is implemented and tested in Section 4.

4 The Solver

4.1 The Basic Recursive Approach

A backtracking algorithm, employing a depth-first approach to examining the search space, is a form of exhaustive search. Pruning conditions that exploit features of the problem domain may be used to reduce time spent examining the search space [9]. The initial exhaustive backtracker is explained first, and effective pruning conditions that reduce the number of states that must be investigated within the search space are incorporated in Section 4.2.

Algorithm 1 Recursive Backtracking Algorithm: Main()

```
Initialise puzzle information, global Iteration_Count
and Solution_Stack.
Current_State becomes the initial_state.
Current_Cell is set to be the first available white cell.
if Solve(Current_State, Current_Cell) is TRUE then
  Print Solution-Stack.
else
  Print "No Solutions".
end if
```

Algorithm 1 calls the boolean Algorithm 2, passing as parameters the initial (empty) grid and a reference to the (first) white cell to be considered. This call is the *initial* call. Algorithm 2 then iteratively attempts to assign values to the white cell passed as a parameter, beginning with the lowest numerical value. An apparently successful assignment of a value to a cell (one which does not violate puzzle constraints) will result in a recursive call to itself, passing as parameters the current partially-filled grid and a reference to the white cell to be considered next. Violations of the puzzle constraints - a duplicate value in a run, an exceeded run-total or an under-target run-total where all possible values have been considered for the final cell of a run - will result in the algorithm returning “False” to the parent. If a solution is reached when the last cell has been considered, the solution is added to the solution stack and “True” is passed to each parent, up to and including that which made the *initial* call, prompting the solution stack to be output. Otherwise, if the largest possible value has been unsuccessfully attempted, “False” is passed to each parent, up to and including that which made the *initial* call, prompting a “No Solutions” message. This approach may be adapted to find all solutions to a puzzle grid, which is useful to determine whether a given puzzle is well-formed. In such a case, when a solution is found and added to the solution stack, the algorithm continues instead of passing “True” to the parent. An iteration count can be added which is incremented each time an attempt is made to assign a value to a cell; this is used as a measure of algorithm performance in Section 5.

Algorithm 2 Recursive Backtracking Algorithm: Solve(Current_State, Current_Cell)

```
for Current_Value from 1 to 9 do
  Increment Iteration_Count.
  Determine runs in which Current_Cell resides, and
  corresponding run-totals.
  Place Current_Value into Current_Cell within Cur-
  rent_State.
  Check resulting Current_State for puzzle violations.
  if [no duplicates in runs] and ([run-total(s) not ex-
  ceeded] or [run(s) completed correctly]) then
    if No White Cells remain then
      Add Current_State to Solution_Stack.
      Return TRUE.
    else
      Current_Cell becomes next available white cell.
      if Solve(Current_State, Current_Cell) is TRUE
      then
        Return TRUE.
      end if
    end if
  else
    Return FALSE.
  end if
end for
```

While this approach is ideal for smaller puzzles, the algorithm may be required to perform a great deal of backtracking in larger puzzles. The algorithm may even reach the final cell before a violation is detected. The addition of further components is desirable, and additional pruning conditions are proposed below.

4.2 Incorporating Puzzle Domain Information

Reducing the number of puzzle states that must be considered may reduce the overall time taken to find a solution. However, the introduction of heuristics and pruning techniques to enable this will come at a cost in processing time. Two approaches to achieving such a reduction are considered here, and evaluated in Section 5.

4.2.1 Projected Run Pruning

The Recursive Backtracking Algorithm 2 of Section 4.1 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 make the run impossible to complete.) In such a case, considerable processing time would be wasted until the Backtracking Algorithm eventually places a sufficiently large value in the initial cell.

Validity checks, termed *Projected Run Pruning*, are added to the Recursive Backtracking Algorithm 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 yields a run-total at least matching the specified run-total for that cell, the backtracker continues, otherwise this fruitless branch of the search space is pruned and backtracking occurs.

Additionally, if only one cell remains unfilled within a run, a check is performed to calculate the difference between the run-totals of both corresponding runs and their current totals. If this difference cannot be met without assigning a value to the remaining cell that is already present in either the horizontal or vertical run in which the cell resides (hence causing a duplication violation), then this fruitless branch of the search space is pruned and backtracking occurs. If a single required value can be placed successfully, it is placed in the cell.

4.2.2 Candidate Set Elimination

In Section 2, each run within a puzzle was assigned a candidate set; this is a set containing all values that can be used to satisfy the given run-total over the given number of cells. Since each cell is a member of two runs, a cell can only contain the values present in the intersection of the two candidate sets that correspond to the two runs in which it belongs. (For example, a run-total of 14 over two cells can be filled using specific pairs of values from the set {5,6,8,9}, and an intersecting two-cell run with run-total 6 has a candidate set {1,2,4,5}; the intersection of these two candidate sets contains only a unique element, 5, that may be placed in the intersecting cell.)

In a *Candidate Set Elimination* cell ordering approach, the checks for Projected Run Pruning are extended so that the "Current_Value" is not assigned to the "Current_Cell" unless it is a member of the intersection of the candidate sets of the two runs in which "Current_Cell" resides. The "Current_Value" is increased until a valid value is reached or the "Current_Value" becomes 9. Also, if a cell has only one value in the intersection of its candidate sets, "Current_Value" now automatically "jumps" to the required value. Hence, many fruitless branches of the search space are pruned due to the absence of certain values in that intersection.

It is expected that Candidate Set Elimination, in combination with Projected Run Pruning will further decrease the number of iterations required to find a solution and hence, the overall solution time required. However, the processing overheads of Candidate Set elimination must not have a detrimental effect on the overall algorithm speed; such overheads may negate any beneficial effects of iteration count reductions.

5 Results and Evaluation

All 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. All times are given in milliseconds.

5.1 The Initial Test Set

Results are shown here for the Recursive Backtracker of Algorithm 2 alone, with Projected Run Pruning (P.R.P.), and with both P.R.P. and Candidate Set Elimination (C.S.E.) added. Few puzzles of small size were available for testing, but a small initial test set was deemed sufficient to examine the methods and to demonstrate both the puzzle-specific nature of Kakuro and the effectiveness of the heuristics. For ten puzzles of each grid size between 2×2 and 10×10 inclusive, the minimum, maximum, median and average solution times, and the median and average numbers of iterations (explained in Section 4.1 required are shown.

The use of P.R.P., and P.R.P. and C.S.E. together, both greatly reduce the iteration count and solution time, demonstrating their effectiveness. As expected, the iteration count when adding C.S.E. always decreases in comparison with P.R.P. alone; impossible assignments to a cell are avoided by examining the contents of the intersected candidate sets that belong to the two runs in

Table 1: Comparative minimum solution times [ms]

		Approach Used		
		Recursive Back-tracker	P.R.P. added	P.R.P and C.S.E. added
Grid Size	2×2	0.96	0.21	4.58
	4×3	1.22	0.55	2.18
	4×4	1.51	0.72	5.88
	5×5	7.67	3.61	7.07
	6×6	5.85	1.51	5.40
	7×7	8.48	3.08	8.72
	8×8	13.43	8.05	12.91
	9×9	216.57	28.21	20.28
	10×10	104.95	24.38	45.14

Table 2: Comparative maximum solution times [ms]

		Approach Used		
		Recursive Back-tracker	P.R.P. added	P.R.P and C.S.E. added
Grid Size	2×2	2.00	0.53	5.16
	4×3	21.30	9.35	11.64
	4×4	105.98	10.64	12.99
	5×5	911.49	150.68	98.03
	6×6	3,454.16	118.48	74.25
	7×7	899.91	180.20	103.07
	8×8	544.27	100.14	69.99
	9×9	33,417.12	163.73	90.33
	10×10	653.00	290.52	223.35

Table 3: Comparative median solution times [ms]

		Approach Used		
		Recursive Back-tracker	P.R.P. added	P.R.P and C.S.E. added
Grid Size	2×2	1.16	0.31	4.89
	4×3	7.51	1.92	6.91
	4×4	7.39	2.09	7.00
	5×5	41.41	10.12	12.44
	6×6	18.22	3.87	10.78
	7×7	29.04	10.63	12.22
	8×8	72.76	36.46	34.07
	9×9	997.16	66.98	51.29
	10×10	352.61	91.35	76.51

Table 4: Comparative Average solution times [ms]

		Approach Used		
		Recursive Back-tracker	P.R.P. added	P.R.P and C.S.E. added
Grid Size	2×2	1.22	0.34	4.87
	4×3	8.80	2.90	7.19
	4×4	21.06	3.86	7.47
	5×5	177.56	47.06	28.17
	6×6	407.54	17.59	17.00
	7×7	243.23	33.90	23.67
	8×8	131.7	42.87	34.58
	9×9	7,963.83	82.80	59.23
	10×10	371.03	106.57	97.63

Table 5: Comparative median iteration counts

		Approach Used		
		Recursive Back-tracker	P.R.P. added	P.R.P and C.S.E. added
Grid Size	2 × 2	56.00	13.00	8.50
	4 × 3	780.00	62.00	24.50
	4 × 4	11,051.00	85.50	26.50
	5 × 5	5,349.00	379.00	224.50
	6 × 6	2,295.00	126.00	51.50
	7 × 7	3,547.00	290.50	133.50
	8 × 8	6,273.00	852.50	573.50
	9 × 9	75,705.00	1,348.00	521.00
	10 × 10	24,646.00	1,590.00	1,010.00

Table 6: Comparative average iteration counts

		Approach Used		
		Recursive Back-tracker	RP.R.P. added	P.R.P and C.S.E. added
Grid Size	2 × 2	55.30	11.20	8.40
	4 × 3	1,325.60	122.30	33.20
	4 × 4	2,717.50	159.40	59.00
	5 × 5	21,449.50	1,767.70	751.50
	6 × 6	56,184.90	577.90	271.70
	7 × 7	23,014.00	943.80	407.70
	8 × 8	11,372.60	1,030.20	571.70
	9 × 9	649,295.30	1,712.70	771.00
	10 × 10	24,059.70	1,849.30	1,364.90

which the cell resides. However, the average solution time decreases only for puzzles of larger size, and by much smaller margins than is evident for iteration count. This suggests that the processing overheads required for C.S.E. cause a significant decrease in speed, negating some or all of the beneficial effects of the reduncion in iteration count. Ultimately, only the overall time taken to obtain to a solution is of importance. More analysis is therefore required to determine whether the additional processing cost of C.S.E. is warranted, when comparing with P.R.P. alone.

5.2 Expanding the Test Set

Initial testing in Section 5.1 makes clear that the addition of P.R.P. greatly reduces solution time, but that further analysis is required to determine whether the addition of C.S.E. is worthwhile. While few puzzles of small sizes are available, a larger number of published puzzles exist for a more "standard" challenge. For an increased test set of puzzles (200 puzzles of grid size 9 × 9, 50 of each other grid size 2 × 2 up to 10 × 10), tests were performed on the most promising methods: Recursion with P.R.P. and also with the addition of C.S.E. This testing attempts to determine whether the latter approach produces consistent and significant reductions in solution time as well as iteration count. Table 8 shows the minimum, maximum, median and average solution times and iteration counts, and the average numbers of iterations per millisecond, for both approaches.

Despite appearing detrimental to the times taken to solve puzzles with smaller grids, the combination of Projected Run Pruning and Candidate Set Elimination is deemed the most successful approach. When compared to P.R.P. alone, C.S.E. successfully decreases the iteration counts for all size groupings of puzzles. The average algorithm speed, indicated by the average number of iterations that can be performed in a millisecond, shows a clear decreasing trend as grid size increased for P.R.P. alone. This trend is with the exception of 2 × 2 grids where puzzle initialisation occupies a larger percentage of the total solution time. The trend is not as evident when C.S.E. is added, possibly due to success of the resultant pruning in greatly reducing the numbers of iterations now required for specific puzzles, and to the additional processing overhead of puzzle initialisation due to cell ordering. That overhead is less evident when larger puzzle grids are solved, which typically require a higher iteration count for their solution. The overheads of generating and intersecting pairs of candidate sets for every cell is clearly more evident in smaller grids because they are more likely to require a small number of iterations for their solution to be found. Table 7 shows how the percentage by which both the median and average iteration counts and solution times for puzzles within this extended test set changed as a result of adding the pruning based on C.S.E. (A negative entry denotes an improvement.)

Table 7: Changes as a result of Candidate Set Elimination (%)

	Iteration Count		Solution Time	
	Median	Average	Median	Average
10 × 10	-37%	-39%	-22%	-31%
9 × 9	-54%	-73%	-48%	-70%
8 × 8	-33%	-48%	-9%	-41%
7 × 7	-55%	-39%	+20%	-38%
6 × 6	-52%	-58%	+18%	-53%
5 × 5	-29%	-70%	+15%	-61%
4 × 4	-70%	-66%	+13%	+3%
4 × 3	-61%	-62%	+352%	+201%
2 × 2	-67%	-32%	+1,666%	+1,610%

6 Conclusions and Future Work

An investigation by the authors into the potential applicability of Kakuro to real-world problems required the development of methods for the efficient automated solution of Kakuro puzzles. The main aim of this paper was to establish whether, and to what extent, puzzle domain information could usefully be employed to speed solution time.

Table 8: Comparative results for the extended test set

	Size	Minimum solution time [ms]	Maximum solution time [ms]	Median solution time [ms]	Average solution time [ms]	Minimum iteration count	Maximum iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
Recursion with P.R.P.	10 × 10	11.33	6,110.72	274.42	637.27	150.00	124,337.00	5,147.50	11,957.06	18.00
	9 × 9	17.13	324,183.89	848.86	11,522.78	331.00	6,517.160.00	17,722.00	240,259.95	20.74
	8 × 8	7.31	4,625.14	52.30	276.97	143.00	102,800.00	1,281.50	6,524.90	23.61
	7 × 7	2.62	147,211.10	9.37	2,972.10	55.00	4,384,770.00	257.00	88,485.62	27.31
	6 × 6	1.51	4,053.05	8.37	215.63	37.00	140,314.00	268.00	7,528.44	31.26
	5 × 5	1.54	2,649.17	11.73	87.21	30.00	95,916.00	415.50	3,291.08	35.72
	4 × 4	0.64	67.24	3.36	8.06	17.00	2,782.00	141.00	346.91	39.61
	4 × 3	0.42	9.35	1.25	1.93	10.00	414.00	52.50	80.94	38.80
	2 × 2	0.18	0.66	0.27	0.28	4.00	13.00	12.00	8.54	30.93

	Size	Minimum solution time [ms]	Maximum solution time [ms]	Median solution time [ms]	Average solution time [ms]	Minimum iteration count	Maximum iteration count	Median iteration count	Average iteration count	Average iterations per millisecond
Recursion, P.R.P. & C.S.E	10 × 10	17.97	3,001.37	213.84	440.10	91.00	52,488.00	3,250.00	7,237.92	14.83
	9 × 9	21.09	64,162.80	438.88	3,482.86	199.00	1,356.402.00	8,160.50	65,350.58	17.30
	8 × 8	12.49	2,265.50	47.85	163.40	101.00	51,876.00	859.00	3,421.68	17.29
	7 × 7	7.29	91,794.53	11.39	1,854.21	31.00	2,694,490.00	115.50	54,187.72	11.94
	6 × 6	6.38	2,179.78	9.92	104.39	21.00	72,375.00	128.00	3,185.70	15.13
	5 × 5	6.26	437.88	14.13	33.88	27.00	13,521.00	296.00	982.56	18.13
	4 × 4	5.35	37.36	6.49	8.28	11.00	1,332.00	42.00	116.44	9.71
	4 × 3	5.11	8.05	5.65	5.81	8.00	115.00	20.50	30.90	5.01
	2 × 2	4.65	5.27	4.77	4.79	4.00	13.00	4.00	5.84	1.22

The nature of the domain information is not compatible with the construction of objective functions in local search approaches, nor for the development of, for example, a fitness function in a genetic algorithm approach. The simplicity of defining a puzzle solution, and the relative ease with which it may be located within its search space leads to the choice of an exhaustive backtracking approach, provided that efficient and effective pruning techniques are suggested by the domain information.

Firstly, Projected Run Pruning was introduced to prune branches of the search space along which a solution cannot lie by considering whether a partially completed run could be validly completed to meet its run totals. Secondly, the size of the intersection of sets of candidate values associated with runs was identified in Section 2 as being important in an understanding of individual puzzle complexity. This measure was used as a cell ordering heuristic (Candidate Set Elimination). For all puzzle sizes, both approaches were shown to greatly reduce the number of iterations required to reach a solution, and the former also produced a marked and consistent decrease in solution time. The processing overheads from adding Candidate Set Elimation increased solution time for smaller puzzle grids, but produced significant reductions for puzzles of sizes that are more typical of those published. Puzzles with large grids typically require a larger number of iterations for their solution, meaning that the benefits of the pruning associated with the candidate sets are more evident. Puzzle domain information has been demonstrated to be useful in significantly reducing solution time.

The effectivess of both heuristics may initially seem to raise the prospect of their usefulness in the solution of very large grids, that may be required in any application of Kakuro to the solution of analagous real-world problems. However, there was a marked decrease in speed as the grid sizes increases. This may be attributed to the fact that larger puzzles typically contain more runs that can be considered ‘long’, greatly increasing the amount of backtracking and the time required to perform the pruning checks. The rate of decrease in speed as grid size increases limits the usefulness of Kakuro in real-world applications that require a mapping to very large grids.

Further analysis of possible arrangements of values in larger Kakuro grids, and the determination of whether all grids of a larger, given size can be fully enumerated, would add to an understanding of puzzle properties. It would also assist in the evaluation of how automated solvers might address the efficient solution of very large puzzle grids.

References

[1] S. K. Jones, P. A. Roach, and S. Perkins, “Construction of heuristics for a search-based approach to solving sudoku,” in *Research and Development in Intelligent Systems XXIV: Proceedings of AI-2007, the Twenty-seventh SGAI International Conference on Artificial Intelligence*, M. Bramer, F. Coenen and M. Petridis, Eds., Springer-Verlag, 2007, pp. 37—49.

[2] G. Galanti, “The history of kakuro,” *Conceptis Puzzles*, 2005. [Online]. Available: http://www.conceptispuzzles.com/articles/kakuro/history.htm.

[3] C. Gomes and D. Shmoys, “The promise of LP to boost CP techniques for combinatorial problems,” in *Proceedings of the Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, N. Jussien and F. Laburthe, Eds., France: CPAIOR, 2002, pp. 291–305.

[4] I. Dotu, A. del Val, and M. Cebrian, “Redundant modeling for the quasigroup completion problem,” in *Lecture Notes in Computer Science*, vol. 2833, F. Rossi, Ed., Berlin: Springer-Verlag, 2003, pp. 288–302.

[5] E. Soedarmandji and R. J. McEliece, “Iterative decoding for Sudoku and Latin square codes.,” in *Forty-Fifth Annual Allerton Conference*, 2007, pp. 488–494.

[6] T. Seta, “The complexities of puzzles Cross Sum and their another solution problems (ASP),” Master’s thesis, University of Tokyo, February 2006.

[7] R. P. Davies, “An investigation into the solution to and evaluation of kakuro puzzles,” MPhil thesis, Faculty of Advanced Technology University of Glamorgan, 2009.

[8] R. P. Grimaldi, *Discrete and Combinatorial Mathematics*, 4th ed. Addison Wesley Longman, 1999.

[9] E. Rich and K. Knight, *Artificial Intelligence*, 2nd ed. Singapore: McGraw-Hill, 1991.

[10] G. F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 5th ed. Addison Wesley, 2005.

[11] L. A. Phillips, S. Perkins, D. H. Smith, and P. A. Roach, “Sudoku and error-correcting codes,” in *Proceedings of the 4th Research Student Workshop, University of Glamorgan*, P. Roach, Ed., 2009, pp. 65-69.