KAKURO SOLVER APPLICATION

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Naomi Takahashi

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

February 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

KAKURO SOLVER APPLICATION

By

NAOMI TAKAHASHI

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

North Dakota State University Libraries Addendum

To protect the privacy of individuals associated with the document, signatures have been removed from the digital version of this document.

ABSTRACT

Takahashi, Naomi, M.S., Department of Computer Science, College of Science & Mathematics, North Dakota State University, February 2010. Kakuro Solver Application. Major Professor: Dr. Kenneth Magel.

In this paper, the application called KakuroSolver is presented that solves Kakuro, the popular logical puzzle that combines Sudoku and crossword. The application accepts a Kakuro puzzle with any size; it then either completely solves the puzzle, or lets the user solve it. This paper discusses the algorithms that can be used to solve Kakuro puzzles as well as other logical puzzles such as Sudoku. Kakuro puzzles are NP-complete, where a search is effective to find solutions. Therefore among those algorithms, KakuroSolver employs the backtracking search along with several techniques that minimizes the search to increase the efficiency. The architecture of the application is also described, and the entire application is evaluated including the future enhancements. The paper also contains the user manual of KakuroSolver.

TABLE OF CONTENTS

ABSTRACT........ iii

LIST OF TABLES........ v

LIST OF FIGURES........ vi

1. INTRODUCTION........ 1

2. RELATED WORK........ 9

3. ARCHITECTURE........ 22

4. ALGORITHM........ 35

5. EVALUATION........ 49

6. USER MANUAL........ 58

BIBLIOGRAPHY........ 70

APPENDIX A. SOURCE CODE – BOARD........ 72

APPENDIX B. SOURCE CODE – CANVAS........ 84

APPENDIX C. SOURCE CODE – CELL........ 88

APPENDIX D. SOURCE CODE – CELLSIZEFORM........ 89

APPENDIX E. SOURCE CODE – CLUECELL........ 91

APPENDIX F. SOURCE CODE – CREATEFORM........ 93

APPENDIX G. SOURCE CODE – INPUTCELL........ 95

APPENDIX H. SOURCE CODE – MAINFORM........ 98

APPENDIX I. SOURCE CODE – TEXTCANVAS........ 108

LIST OF TABLES

Table	Page
1. Possible Combinations of Numbers	5
2. Syntax of Puzzle in a Text File Format	61

LIST OF FIGURES

Figure Page

1. Kakuro Puzzle and Solution 2
2. Kakuro Puzzle Example 1 3
3. Solution for Kakuro Example 1 3
4. Kakuro Puzzle Example 2 6
5. Kakuro Puzzle Example 3 7
6. Kakuro Puzzle Example 4 8
7. Pseudocode of Rule-based Approach 14
8. High-level Class Diagram of KakuroSolver 23
9. Sequence Diagram: Create a Puzzle 28
10. Sequence Diagram: Solve a Puzzle 30
11. Sequence Diagram: Check a Puzzle 32
12. Sequence Diagram: KeyPress on Canvas 33
13. Pseudocode of the *solve* Method 39
14. Order in which Cells are Visited 41
15. Kakuro Puzzle with Multiple Solutions 51
16. Screenshot: *check* Function Returning Incorrect Result 53
17. Screenshot: Error Message regarding .Net Framework 3.5 54
18. Screenshot: Initial Screen (Main Window) 58
19. Screenshot: Specify the Size of Puzzle 60
20. Screenshot: Create a Puzzle 60

21. Example of Puzzle in a Text File Format.. 62

22. Screenshot: Example of Puzzle Displayed in the Main Window...................... 62

23. Screenshot: Open a File containing a Puzzle.. 63

24. Screenshot: Puzzle Solved by the KakuroSolver.. 64

25. Screenshot: Select an Input Cell in the Puzzle... 65

26. Screenshot: Enter a Number in the Selected Input Cell.................................... 65

27. Screenshot: Enter a Choice in the Selected Cell... 66

28. Screenshot: Remove a Choice from the Selected Cell...................................... 67

29. Screenshot: Message Displayed when All User Inputs are Correct.................. 67

30. Screenshot: Message Displayed when There are Incorrect Numbers............... 67

31. Screenshot: Save a Puzzle in a Text File.. 69

32. Screenshot: Exit the KakuroSolver.. 69

1. INTRODUCTION

Kakuro, also known as Cross Sums, is a logical puzzle that combines Sudoku and crossword. A player is supposed to fill all the white empty cells (input cells) with numbers 1-9, so that the sum of numbers entered in a run equals the corresponding clues, which are small numbers shown in the black cells.

Same as Sudoku, Kakuro is considered as a constraint satisfaction problem, which is based upon "predefined, explicitly given constraints" [2]. It is too complex for a human player to solve with a simple trial-and-error approach. Search algorithms should be able to solve a Kakuro puzzle more efficiently, as well as logical reasoning.

This project aims at designing and developing a desktop application called KakuroSolver, which is capable of solving Kakuro puzzles with any size and any difficulty. The application should have two modes:

(1) Completely solves the puzzle

(2) Lets the user solve the puzzle, by maintaining a list of possible choices (1-9) for each input cell in the puzzle

In either mode, the application should allow the user to create a new puzzle or to open a saved puzzle in the text file format. Mode (1) should fully solve the puzzle and display the solution, and mode (2) should allow the user to enter numbers in the input cells, either as a final or one of the possible choices. As part of mode (2), the user should also be able to try to solve the puzzle after entering some numbers and to check whether the inputs to the point are correct.

The rest of this section briefly explains Kakuro rules and some techniques that are generally used by human players to solve the puzzle.

1.1. Kakuro Rules

A Kakuro puzzle is a grid with white cells and black cells. Figure 1 shows a typical Kakuro puzzle with the solution. White cells are called input cells, where players can fill with numbers from 1 to 9. Some black cells are clue cells, which have a slash from top-left to bottom-right and with one or two numbers. The number on the top-right of a clue cell is the across clue, which represents the sum of the corresponding across run; the one on the bottom-left is the down clue, which represents the sum of the corresponding down run. Pure black cells are called blank cells, which are used to separate runs on the same row or column and to fill the empty space on the puzzle. Similar to Sudoku, it is not allowed to enter the same number more than once in a run. However, the same number can be entered multiple times in a row or column, if the numbers are in different runs and are separated by blank cells.

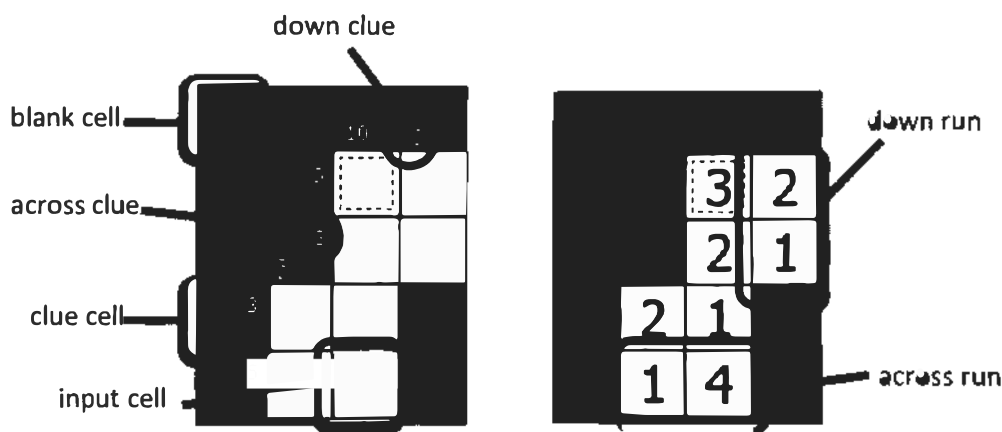


Figure 1. Kakuro Puzzle and Solution [10]

In the Kakuro puzzle shown in Figure 2, there are four input cells (B2, B3, C2, and C3) and three clue cells (A3, B1, and C1). The cell B1 has a down clue of 5, which is the sum of two input cells, B2 and B3. Similarly, the sum of C2 and C3 has to equal the down clue 16 in C1, and the sum of B3 and C3 must be equal to the across clue 12 in A3.

	A	B	C
1		5	16
2			
3	12		

Figure 2. Kakuro Puzzle Example 1

In other words, we can have the following relationships in the four input cells (B2, B3, C2, and C3):

(1) B2 + B3 = 5

(2) C2 + C3 = 16

(3) B3 + C3 = 12

Based on these relationships, the numbers that may possibly be entered in each cell can be identified. For example, the possible combinations of numbers for B2 and B3 are (1, 4), (2, 3), (3, 2), and (4, 1). In the same way, the possible combinations for C2 and C3 are (7, 9) and (9, 7), as the combination (8, 8) is not allowed. Also possible combinations for B3 and C3 are (3, 9), (4, 8), (5, 7), (7, 5), (8, 4), and (9, 3).

From the relationship (3), C3 can be 3, 4, 5, 7, 8, or 9. The relationship (2), however, shows that it can be only 7 or 9. This fact narrows down the possible combinations for B3 and C3 to (3, 9) or (5, 7), and hence B3 can be either 3 or 5. From the relationship (1), B3 cannot be 5; therefore, only 3 can be entered in B3.

Once 3 is entered in B3, the numbers for other cells are determined. B2 is 2, and C3 is 9; therefore, 7 should be entered in C2 (see Figure 3).

	A	B	C
1		5	16
2		2	7
3	12	3	9

Figure 3. Solution for Kakuro Example 1

1.2. How Human Solves Kakuro

To solve a Kakuro puzzle, human players need to identify possible combinations of numbers for each run and find candidate numbers possibly entered in each input cell. There are several techniques that human players may generally use to solve a Kakuro puzzle.

1.2.1. Combination of Numbers

In general, players start with a clue that has the least possible combination of numbers, usually those with two or three input cells. For example, a clue of 3 over a run of two input cells has only one possible combination of 1 + 2, where a clue of 12 over four cells has two possible combinations, 1 + 2 + 3 + 6 and 1 + 2 + 4 + 5. Table 1 shows all the possible combinations of the size of runs and the clues, where there is only one combination of numbers.

1.2.2. Isolated Cells

When all the other cells in the same run are filled, the remaining cell's value can be calculated by simply subtracting the sum of the values of the other cells from the clue. For example, three of the four input cells with the clue of 14 are filled with 1, 3, and 6, the value of the remaining cell should be 4 (= 14 – (1 + 3 + 6)).

1.2.3. Largest/Smallest Possible Numbers

The clue for a run often has more than two possible combinations of numbers. If a clue involves more than two possible combinations, it may not be efficient to try every combination one by one. In such cases, players may identify the largest or smallest possible number for the run to reduce the number of possible combinations.

Table 1: Possible Combinations of Numbers

Size of run	Clue (Sum)	Combination of numbers
2	3	1 + 2
	4	1 + 3
	16	7 + 9
	18	8 + 9
3	6	1 + 2 + 3
	7	1 + 2 + 4
	23	6 + 8 + 9
	24	7 + 8 + 9
4	10	1 + 2 + 3 + 4
	11	1 + 2 + 3 + 5
	29	5 + 7 + 8 + 9
	30	6 + 7 + 8 + 9
5	15	1 + 2 + 3 + 4 + 5
	16	1 + 2 + 3 + 4 + 6
	34	4 + 6 + 7 + 8 + 9
	35	5 + 6 + 7 + 8 + 9
6	21	1 + 2 + 3 + 4 + 5 + 6
	22	1 + 2 + 3 + 4 + 5 + 7
	38	3 + 5 + 6 + 7 + 8 + 9
	39	4 + 5 + 6 + 7 + 8 + 9
7	28	1 + 2 + 3 + 4 + 5 + 6 + 7
	29	1 + 2 + 3 + 4 + 5 + 6 + 8
	41	2 + 4 + 5 + 6 + 7 + 8 + 9
	42	3 + 4 + 5 + 6 + 7 + 8 + 9
8	36	1 + 2 + 3 + 4 + 5 + 6 + 7 + 8
	37	1 + 2 + 3 + 4 + 5 + 6 + 7 + 9
	38	1 + 2 + 3 + 4 + 5 + 6 + 8 + 9
	39	1 + 2 + 3 + 4 + 5 + 7 + 8 + 9
	40	1 + 2 + 3 + 4 + 6 + 7 + 8 + 9
	41	1 + 2 + 3 + 5 + 6 + 7 + 8 + 9
	42	1 + 2 + 4 + 5 + 6 + 7 + 8 + 9
	43	1 + 3 + 4 + 5 + 6 + 7 + 8 + 9
	44	2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
9	45	1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9

In the Kakuro puzzle shown in Figure 4 below, if 1, 2, and 3 are entered in the first three input cells for the down run, the value of the fourth (last) input cell must be 6 (= 12 – (1 + 2 + 3)). The number 6, therefore, would be the largest possible number for

any input cell in this run. In other words, any larger number (7, 8, or 9) cannot be entered in this run. This strategy of determining the largest possible number for a run actually helps to decrease the number of possible combinations for the across run. The across run has five possible combinations (1 + 2 + 8, 1 + 3 + 7, 1 + 4 + 6, 2 + 3 + 6, 2 + 4 + 5) for a clue of 11 over three cells. Yet the first two combinations are impossible in this case, as they contain the number which is larger than 6.

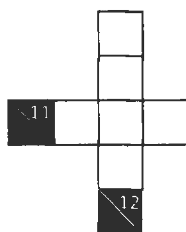


Figure 4. Kakuro Puzzle Example 2

This method of identifying the maximum possible number is called the *minimax* method [2]. In the same way, the smallest possible number can also be identified. Assuming that all the input cells except for one are filled with largest numbers starting with 9, 8, 7, and so on, the number for the last cell can then be identified. That number is the minimum possible number for the run.

These techniques are not always available for a run. For instance, the minimum possible number cannot be identified in the example 2, because entering 9, 8, and 7 in the first three input cells for the down run identifies the negative value (-12 = 12 – (9 + 8 + 7)) as the minimum possible number. Correspondingly, when the clue for a run is large, the maximum possible number could be greater than 9, which is an invalid number in Kakuro. Yet identifying the maximum and/or minimum possible number for a run will limit the possible combinations, whenever it is available.

1.2.4. Cross Reference [9]

In two intersecting runs, the intersection cell's value may be identified from the possible combinations for the two runs. In the example shown in Figure 5, the across clue of 20 has four possible combinations: 3 + 8 + 9, 4 + 7 + 9, 5 + 6 + 9, and 5 + 7 + 8. However, there is only one possible combination for the down run: 1 + 2 + 3. This indicates that only the 3 + 8 + 9 combination is available for the across run, since it is the only combination that contains 1, 2, or 3. Further, it is revealed that the intersection input cell's value is 3. By viewing the common number in the possible combinations for two intersecting runs can narrow down the possible combinations of these runs, and/or even determine the value of the intersection input cell.

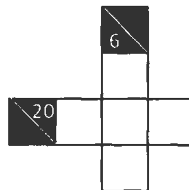


Figure 5. Kakuro Puzzle Example 3

1.2.5. Combo Reference [9]

The possible combinations for a run may be narrowed down until only one is left, by doing a cross reference on every input cell in the run. In the example shown in Figure 2, the across run has the clue of 12, where three combinations are possible (3 + 9, 4 + 8, and 5 + 7). The left down run has two possible combinations (1 + 4 and 2 + 3), and thus the 5 + 7 combination can be eliminated for the across run, because neither 5 nor 7 can be entered in the intersection input cell. Equally for the right down run, there is only one combination, 7 + 9, which eliminates the 4 + 8 combination for the across run, as neither 4 nor 8 can be entered in the intersection input cell.

Therefore, only one combination (3 + 9) is left for the across run. This determines the values of the two intersection cells (3 for the left input cell and 9 for the right input cell), which then determines the values of the remaining cells: 2 for the top-left cell and 7 for the top-right input cell.

1.2.6. Filled Areas [9]

The value of the input cell outside of the empty *n* x *n* area may be deduced by identifying the sum of cells in the *n* x *n* area. In Figure 6, the sum of input cells in the empty 2 x 2 area should be 7, which is the sum of the two across clues, 4 and 3. However, the sum of two down clues is 10 (= 4 + 6). The difference between these two numbers, which is 3, should be the value of the cell outside of the 2 x 2 area (C1).

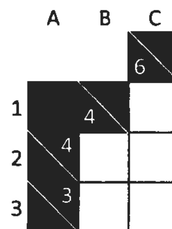


Figure 6. Kakuro Puzzle Example 4

2. RELATED WORK

Numerous approaches and techniques have been proposed to solve logical puzzles, especially Sudoku. Also there are plentiful Kakuro solver applications available online, some of which provide the source code or at least the algorithm used in the application. This chapter consists of two sections: "Logical Puzzle Solving Algorithms" and "Existing Kakuro Solvers". The "Logical Puzzle Solving Algorithms" section discusses techniques used to solve logical puzzles in general, and those techniques should also be applied in a Kakuro solver application. The "Existing Kakuro Solvers" section describes several Kakuro solving algorithms. Some of them are actually employed in a Kakuro solver application, and others are theories that may be utilized in a Kakuro solver application.

2.1. Logical Puzzle Solving Algorithms

2.1.1. Search Algorithms

[22] proves that Kakuro, not only Sudoku, is NP-complete. A number of metaheuristic approaches have been examined to solve an NP-complete problem, and one of the popular ways is a search. This paper discusses three major search algorithms: backtracking search, local search, and Dijkstra's Algorithm.

2.1.1.1. Backtracking Search

The simplest approach to find a solution is a sole depth-first search, which is also called a brute-force search. It considers "no information other than the initial state, the operators of the space, and a test for a solution" [12], and hence traverses every possible path to the end whether or not it is valid. It is time-consuming but guarantees that a solution is found [4].

The backtracking search also employs a depth-first search, but has better performance than the brute-force search. In this approach, an empty cell is picked and a number is entered to see whether it is conflicting with other cells. If there is a conflict, the process is reattempted with the next number. If there is no conflict, the number fills the empty cell, and another empty cell is chosen to repeat the same process. If an empty cell turns out to have no possible number, the process will go back to the previously filled cell, and another possible number will be entered in the cell.

This backtracking method is still a slow approach if it is used alone, because in the worst case, nine candidate numbers have to be examined for a cell. To increase the efficiency, this method should be used along with some other techniques.

The first technique is *candidate elimination* [4]. In Kakuro, every input cell always has nine candidates (1-9) in the beginning. A candidate for a cell is eliminated when it has been already entered in another cell in the same run (i.e., row, column, or box if the puzzle is Sudoku).

Actually the first step of the backtracking search is nothing but a type of candidate elimination, but this method can do more. In Kakuro, not all the runs have the same number of cells. Candidates for the input cells in a run can be identified from the number of cells in the run and the clue number for the run. Since an input cell may belong to two runs (across and down), the number of candidates for the cell can sometimes be reduced to only one. This means the candidate is the numbers that should be actually entered in the cell.

If the number of candidates is more than one, then one of the candidates will be tested via the backtracking search. If it is successful, the choice is correct. If it is unsuccessful, the choice is incorrect and one of the other candidates will be tested.

The next technique to improve the backtracking approach is *forward checking* [17], which maintains a list of candidates for each input cell. In the original backtracking approach, candidates for a cell are identified when the cell is picked. It is, however, more efficient to create and maintain a list of candidates for each input cell, and remove a candidate from the list whenever the number is entered in another cell in the same run. Then when the cell is picked, there is no need to go over the cells in the same run; only the numbers in the candidates list should be considered.

The third technique is called *constraint propagation* [17], which is further discussed later in 2.1.2. Whenever a number is entered in a cell, the possible numbers are recomputed for all the dependent cells, which are the ones in the same run as the selected cell. If a cell with no possible number is found, the solution fails and the process will return to the cell previously filled.

The fourth technique is *minimum remaining values heuristic* [17]. This decides the order in which cells are selected and guessed, so that the size of the entire search tree is minimized, reducing the number of branches at each level. Instead of simply selecting the first empty cell, the empty cell with the least candidates is chosen. It becomes more efficient if it is used with the other techniques discussed above, as the candidates are updated in each iteration.

2.1.1.2.Local Search

A local search is based upon complete assignments. In this type of search, some value is assigned to each variable, and "the assignment is iteratively *repaired* until a solution is found" [21].

One example of this local search is a *steepest ascent hill climbing* [17], which iteratively improves the selected assignment by comparing all successors to identify the closest one to the solution. It has a drawback that the search may reach a local minimum. An easy way to assess this issue is to employ *random-restarting hill climbing* [17], also known as *Shotgun climbing*. It repeats hill climbing, each time with a random assignment, so that local minima may be avoided and the search can be carried on.

Another local search that can handle local minima is a *tabu search* [17]. It keeps past repairs in a list called a tabu list, so that it can avoid visiting the repairs again which lead to the assignment that has already been visited. The size of a tabu list is usually limited, and older repairs are removed as new repairs are added. This way, a large number of repairs can be executed in a single iteration. Tabu lists are efficient, but it is always possible that a recent repair may lead to a new better assignment, because changing the value of a node places the node in the tabu list. Therefore, *aspiration criteria* are generally employed; if an aspiration criterion is met, it is allowed to select a repair in the tabu list.

2.1.1.3. Dijkstra's Algorithm

Dijkstra's Algorithm [5] is generally used in a weighted directed graph to identify a shortest path from the initial node to all the other nodes, producing a minimum spanning tree. This algorithm, however, can be easily altered to "find a minimum cost path from

the start node to the goal node, that is, an optimal path" [7], without having to visit all the nodes. [7] introduces this as Generalized Dijkstra's Algorithm.

The Generalized Dijkstra's Algorithm contains two lists of nodes: CLOSED and OPEN. The CLOSED list maintains the set of nodes that were visited and expanded; the OPEN list has the set of nodes that were already visited but not expanded yet. Starting from the initial node, the algorithm visits its neighbor nodes (successors), keeps them in the OPEN list. It then selects the node with smallest weight in the OPEN list to expand it. The selected node is then moved to the CLOSED list, and its immediate successors are inserted into the list OPEN. The algorithm iterates this process, selecting the node with minimum weight from the OPEN list. When it reaches the goal node for expansion, an optimal path is found.

In Kakuro, the weight assigned to each node can be the number of candidates. In this case, the Generalized Dijkstra's Algorithm can be seen as a backtracking search with minimum remaining values heuristic, because this backtracking search always changes the order in which the remaining nodes are visited, so that the node with least candidates is visited first.

2.1.2. Constraint Propagation

Another approach to solve logical puzzles is a rule-based approach, or constraint propagation. It is close to the way a human player generally solves a puzzle. The pseudocode of this approach is shown in Figure 7. [19] discusses how to solve a Sudoku puzzle as a constraint problem.

The goal of this approach is to solve a puzzle without guessing. A well-defined puzzle should be free from guessing, which means that by reasoning the candidates for at

least one cell should be always limited to only one in the end, if more than enough constraints are identified. The redundant constraints technique, which combines multiple original constraints, may be employed in order to reinforce the deduction of candidates. The rules utilized may be techniques used along with a backtracking search or techniques used by human players, such as the ones discussed in 1.2.

```
Begin
  While solution Not found
    If Rule1
      Do Action1
    Else If Rule2
      Do Action2
    Else If Rule3
      Do Action3
       :
       :
    Else
      Return false
  End While
  Return true
End
```

Figure 7. Pseudocode of Rule-based Approach

2.2.Existing Kakuro Solvers

In this paper, seven Kakuro solving programs are examined. All of them employ the backtracking search algorithm, though every program utilizes different techniques to identify candidates for a cell.

2.2.1. OddThinking [1]

The blog naming OddThinking introduces a Kakuro solver application written in Python. Although the actual source code is not available, the algorithm used in the application is described in the blog. The application employs numerous "what if" investigations: "If this cell was the maximum it could be, and this cell was the maximum it could be, then that would dictate the minimum value the last cell could be" [1].

In the application, the main program constructs the Grid, in which there are the Cells constructed by the Grid. In the Grid, there are also the Rows. A Row represents either a down or across run, and a set of Cells are associated with the Row. Each Row has the PossibleRowValues, an array of possible numbers for the run; therefore, a PossibleRowValues may have up to nine numbers from 1 to 9. When there are multiple possible combinations for a Row, each combination makes a PossibleRowValues. It means that each Cell may be associated with one or more across PossibleRowValues and one or more down PossibleRowValues.

The possible values for a Cell then are determined by comparing the down and across PossibleRowValues. Whenever a possible number is eliminated from a PossibleRowValues, it will be notified to all the Cells associated with it, so that the possible values for the Cells can be narrowed. This is actually a recursive algorithm. When a number is eliminated from one of the PossibleRowValues, all the Cells in the Grid reconsider its possible values.

This application was tested with an empty novice-level puzzle and a half-solved Ninja-level puzzle, both of which it solved successfully according to the author.

2.2.2. SAAM's [18]

SAAM's provides a Kakuro solver application written in Java with Applet. This application goes over the board, and for each clue, it determines the cells in the same run and the possible combinations of numbers. For each cell, all the possible numbers are then identified by looking at the cell's across clue and down clue.

For example, if there are four possible combinations for a cell's across run (1 + 5, 2 + 4, 4 + 2, and 5 + 1), the possible numbers for the cell would be 1, 2, 4, and 5. If there

are two possible combinations for the same cell's down run (1 + 3 and 3 + 1), the possible numbers for the cell would be 1 and 3. By comparing these two lists, the possible numbers for the cell can be limited to 1, because this number is the only one common number in the two lists.

Whenever only one number is left for a cell, the number is set to the cell, and all the combinations that do not contain the number for the cell will be deleted from the possible combinations list. The application keeps looping these steps until no more changes can be made, assuming that the board is not filled yet. The application then starts guessing. For a cell, one of the possible numbers is chosen and the application tries to fill the other cells for the case(s) when the cell is filled with the number. When any error or inconsistency is found, the selected number is turned out to be inappropriate for the cell. The combinations containing the number will be deleted from the possible combinations list, and another number is chosen from the remaining combinations. This process is repeated until all the cells in the board are filled without any error.

2.2.3. Meilof [20]

Meilof Veeningen et al. have developed a Kakuro solver application written in Java. They use the combination of *simple iterations* and *level 1-iterations* [20]. In a simple iteration, each combination for a run is checked whether it is in harmony with the other information (other clues and cells) thus far, based on the cross reference technique. Then for each cell in the run, the possible numbers for the run are set to be the possible numbers for the cell.

In the level 1-iteration, the application picks a run, and tries the cells in the run with one of its possible combinations, to investigate what happens when simple iteration

is used on this. In most of the cases, all but one combination will turn out to be conflicting, so that the possible combination can be limited to one.

The application first tries the simple iteration until no more results are returned, and then tries one step of level 1-iteration to see what the simple iteration would give again. As such, the simple iteration and the level 1-iteration will be tried in turn, until the entire puzzle is solved or the level 1-iteration returns no more results.

In addition, this application employs a priority queue to manage all the runs in the puzzle. The queue is sorted by the number of possible combinations associated with the run. The runs with less number of possible combinations will have higher priority. Whenever all the possibilities for a run are tried, the run should be removed from the queue.

2.2.4. Ben-Haim [3]

Ben-Haim has developed a Kakuro solver application in C#. The Board object contains a two-dimensional array of Elements. An Element represents individual cell in the Board, and can be either a value element or a sum element. When an Element's *value* variable contains a number from 1-9, the Element is considered as a value element. When an Element has number(s) in either *sumDown* variable, *sumRight* variable, or both, it is considered as a sum element.

This application uses the recursive algorithm. First of all, it identifies all the possible numbers for each value element (cell). The application will delete any numbers that are already entered in the across run or down run, because the puzzle does not allow the user to use the same number twice in the value element. In the next step, it calculates the minimum and maximum possible sum for the other value elements that are still empty

in each run (excluding the current element). Then the illegal numbers can be identified where

- (sum of values already entered in the run) + (minimum possible sum for the other empty value elements) + (the illegal number for the current value element) > (sum of values for the entire run)

or

- (sum of values already entered in the run) + (maximum possible sum for the other empty value elements) + (the illegal number for the current value element) < (sum of values for the entire run).

Again the application does this step for both the down run and the across run of a value element to identify the illegal values for a value element. The numbers from 1-9 that are not illegal should then be the possible numbers for the value element.

Next, the element that has the least possible numbers is chosen as the best element, which the application starts working on. For that element, the application tries to fill one of the possible numbers, followed by the first step, in which the second best element should be chosen. These steps continue recursively until the last value element is reached. When an error occurs during the process, the number last entered will be considered to be illegal, and the next possible number for the element will be tried.

2.2.5. Alan Lerner [13]

Alan Lerner briefly provides the algorithm to solve Kakuro puzzles as part of the course he has taught. This algorithm employs the recursion. In the step 1, all the squares in the board (puzzle) are visited. If there is any square (input cell) that has only one option (possible number entered), the option is assigned to the square. The assigned

number then will be removed from the list of options for the empty squares along the down run and the across run the square belongs to. In addition, the value is deducted from the clues for those runs.

In the step 2, again all the squares in the puzzle are visited, and clues are taken. For each clue, the minimum and maximum possible values are identified. Based on these numbers, the limits are set to all the squares that belong to the run. If the number of empty squares is two and the clue is an even number, the average number of the clue is also eliminated from the list of options.

These two steps are looped until no more changes can be made to the board. The status of the board is then checked, because the entire board may be solved during the loop of the step 1 and 2, or any inconsistency may be found during the loop, which indicates the board has no solution.

If the board is still unsolved after the loop of the step 1 and 2, the copy of the board to this point is created. The process then identifies a square which is still empty, and it guesses a solution for the square from its option. Subsequently an option for the square is set, and the option is removed from the list of options for the square in the copied board. With the option in the square, the rest of the board tries to be solved by repeating from the step 1. If the current board has no solution with the option in the square, then the application attempts to solve the copied board next. The option set in the board is not included in the copied board, allowing the next option to be examined on the copied board. This recursion is continued until the board is solved or found to have no solution.

2.2.6. Kakuro Solver

The blog Kakuro Solver [11] discusses how to implement a Kakuro solver programmatically. The algorithm consists of four steps. In the step 1, all empty white cells (input cells) are visited. For each empty white cell, all the possible values for the cell are identified. There is the hash table available, which contains all the combinations of possible values, along with the number of empty cells in a run and the clue for the run. From a clue and the number of white cells that belong to the run, the possible numbers can be identified by using this hash table.

After that, the existing values in the run are removed from the list of possible values. At the same time, the combinations of possible values that do not contain the existing values must be also removed, because those combinations are inconsistent with the existing values.

This sequence of processes are done for both the down run and the across run that the empty white cell belong to. In the step 2, the possible values for the down run and those for the across run are intersected. As the empty cell belongs to both runs, it should contain one of the values that are common in both lists. If there is only one value, the value is placed in the cell.

The step 3 loops the step 1 and 2 until there is no empty white cell that has only one possible value. Then in the step 4, the empty cell with the least possible values is selected. All possible values for the cell are placed in the cell one by one for trial. Every time a possible value is placed, the steps 1 through 4 are repeated for the remaining empty cells, until all the remaining cells are filled with values successfully. If any cell that has

no possible value is found instead during the iteration, the possible value is not correct for the cell, and thus the next possible value is placed for trial.

2.2.7. k4kur0 [6]

The k4kur0 is a freeware written in Java, which is designed for mobile phones or PDAs. In this application, all the sums (runs) are visited first. For every sum, every possible combination for the clue value is iterated to evaluate whether it can be applied to the fields (cells) regarding the current set candidates. If a combination of candidates matches to the current set candidates, it is stored to a new candidates set. After all combinations are iterated over, the current candidates are replaced with the new candidates. If a field now has only one candidate left, the field is stored as solved, and the value is removed from the candidates within the remaining fields of the relevant row and/or column.

This sequence of processes is repeated until the entire puzzle is solved, or no more changes can be made. If the puzzle is not solved yet but no more change can be made in the next iteration, the iteration should be terminated and the guessing should start. A field with least number of candidates is chosen, and the candidates are iterated over, assuming that a candidate is the solution. With this assumption, the rest of the puzzle is solved. If the solving is successful, the solution is stored and the next candidate is tried. If no other solution is found, the puzzle has only one solution. If any other solution is found, the solving is terminated because the puzzle is not well-formed, having multiple solutions.

3. ARCHITECTURE

This section describes the architecture of KakuroSolver. Taking into account the easiness of designing and maintaining the graphical user interface, Visual Studio 2008 was selected as the integrated development environment. As the programming language, C# was chosen because the author had the experience with C# most in the languages available in Visual Studio.

KakuroSolver requires heavy interaction with the user, and visual graphics are necessary to properly display a puzzle. The Windows Forms are considered to be appropriate to implement these over the console window. The application was, therefore, developed as the Windows Forms application. Based on this, all objects needed in the application were identified in order to perform the required tasks.

Figure 8 shows the high-level class diagram of the application. The Windows Forms created in the application are as follows: MainForm, CreateForm, and CellSizeForm. The MainForm has a Canvas and a Board, and it opens the CellSizeForm. The CellSizeForm then opens the CreateForm, which has a TextCanvas and a Board. Canvas and its child TextCanvas both have a Board. Board has Cell, which is inherited by ClueCell and InputCell.

All the classes were identified as follows. First of all, a Windows form called *MainForm* must exist. It is responsible for the tasks related to displaying of the puzzle. It also needs to have some controls (i.e., menus and buttons) for the user to initialize the action of interest, such as creating a puzzle and solving a puzzle. In addition, there should be an object that actually makes the puzzle visible to the user and tracks user

inputs. It could be a type of two-dimensional arrays, since the puzzle can be any size and need some flexibility in size.

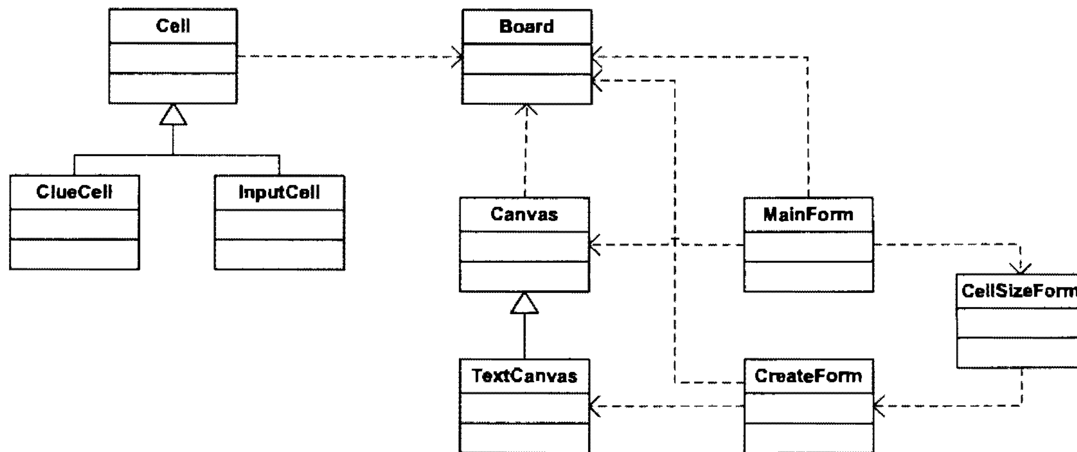


Figure 8. High-level Class Diagram of KakuroSolver

A two-dimensional TextBox array was first considered, because user inputs should be handled easily with TextBox objects. Actually [3] employs TextBox objects to represent individual cells. A TextBox is, however, designed to be a rectangle with by far more width than height. To display a Kakuro puzzle, it is ideal to make each cell a square, but it is difficult to set a TextBox's height in C#. The height of a TextBox is determined by the following calculation, where FontSize is the size of the font, FontLineSpacing is the distance between two lines of text in design units, and FontEmHeight is the height of the widest letter's font in design units, normally the letter "M":

Height = (FontSize * FontLineSpacing / FontEmHeight) + 7 [8].

This means that the height of a TextBox is purely dependent on the font size, and identifying the font size for a TextBox with the ideal height is the only way to specify the height of the TextBox. Even though the ideal height is set by specifying the font size, the TextBox then may contain at most two digits if the width of the TextBox is the same as the height. Consequently, it is impossible to make the TextBox a square in order to draw

a clue cell, which may contain up to five characters: a two-digit down clue, a back-slash, and a two-digit across clue.

Therefore, a user control called *Canvas* was defined for the display of the puzzle. Resembling Java Applet, any objects that System.Drawing.Graphics supports (i.e., lines, rectangles, and texts) can be drawn on the Canvas. Thus clue cells can be drawn more adequately. Moreover, Canvas allows drawing numbers at any font size and at any place, which is more suitable to display the possible choices for input cells. Each number entry needs to be handled in a KeyPress event or a KeyDown event. It is, however, generally efficient enough to use the Canvas, since the number the user enters in the puzzle at a time is always one-digit (1-9).

Next, the Cell class was identified, where a two-dimensional Cell array makes the actual puzzle. This Cell array is used to maintain user inputs and to solve the puzzle. Although the Cell array could be placed in the MainForm along with the *solve* function, it would complicate the MainForm and make the future changes more difficult. The *Board* class was, therefore, determined as the mediator of the application, which lowers the coupling of classes.

With these classes, all the tasks except for creating a puzzle should be handled. The application should clearly differentiate the creation of a puzzle from the other user actions such as solving the puzzle or saving the puzzle, which are available when the puzzle is loaded in the application. Not to confuse the user which actions are currently available, the *CreateForm* was designed only to create a puzzle. Also another Windows Form called *CellSizeForm* was created for the user to enter the width and height of the puzzle, which should be displayed before the CreateForm is displayed.

On the CreateForm, the user will specify what each cell should be: An input cell, a clue cell, or a blank cell. If a clue cell is specified by the user, the down and/or across clue, which is a one- or two-digit number, should be also designated. Accordingly the CreateForm must have a grid, and the user must be able to type the numbers and/or back-slashes in each cell. The Canvas was first considered to represent the grid, but there were several drawbacks found.

When this grid is drawn on the Canvas, each number entry must be recognized by the application as a numeric key is pressed and released. This feature would make it difficult for the user to type two-digit numbers in a clue cell smoothly. For example, when the user enters "12" by mistake and wants to delete "2", the program should simply allow the user to correct it by pressing the Backspace key. Handling this sequence of actions as key press events (pressing "1", "2", and then Backspace) would be inefficient, since the Canvas is required to be refreshed (repainted) whenever each key is pressed. It might not be able to react fast typing.

Another option to handle the creation of puzzles is making the user to specify the number in a clue cell from a list of number selections. The list would display the numbers from 3 to 45, which are the smallest and the largest possible clue respectively. In this way the user clicks and selects the number from the list, and then clicks the cell to enter the selected number in the cell as either a down clue or an across clue. This way of input is employed in many Kakuro applications that allow the user to create a puzzle. However, it would require countless mouse movements and clicks.

Therefore, the *TextCanvas* class was identified to minimize the number of user actions compulsory to specify each cell, as well as to draw the grid adequately. This class

inherits the Canvas because there are several functions common to the Canvas and the TextCanvas. These common functions include drawing the frame of a grid and determining the size of each cell of the grid based on the puzzle size. Unlike the Canvas, the TextCanvas adds a TextBox in each cell of the grid, so the user can easily type in the cell and modify what is entered in the cell. Though the height of a cell is larger than the height of the corresponding TextBox, the TextBox should get the focus when the user clicks anywhere on the cell.

Besides these classes, two more classes were added while the puzzle solving algorithm was being developed: ClueCell and InputCell. Both classes are the children of the Cell class, and they represent a clue cell and an input cell respectively. Initially there was only the Cell class. The clue cells and the input cells were both Cell objects, and they were differentiated based upon which variables of Cell are initialized plus what string value was stored in the *cellType* variable. However, as more variables became required only for one of these two types, Cell must have more number of variables in total, with half of them being unused. Moreover, the number of methods also increased as both cell types needed many methods, most of which are mutually exclusive. It was then concluded that creating two child classes of Cell should solve this complexity.

The rest of this section discusses how these classes interact with each other to perform a specific task.

3.1. Create/Open a Puzzle

Figure 9 is the sequence diagram for creating a puzzle through the CreateForm. When the user calls the *createPuzzle* method by either clicking the Create button or selecting the Create option on the menu, the MainForm first checks whether these is a

puzzle currently loaded. If there is, the MainForm asks the user if he/she wants to save the current puzzle, and saves it as a text file in the location of the user's choice. It then creates the CellSizeForm in which the user specifies the size of the puzzle, while disabling itself so that the user is not confused with other actions that are currently unavailable. On the CellSizeForm, the user has two options: To create a puzzle or to cancel and go back to the MainForm. When the user selects Cancel, the CellSizeForm closes itself and enables the MainForm. When the user selects Create, it examines whether the user enters the valid numbers (positive integers) as the puzzle's width and height. If either or both inputs are invalid, the error message should be displayed as a MessageBox; otherwise the CellSizeForm closes itself and creates the CreateForm.

When the CreateForm is created and loaded, it creates a two-dimensional TextBox array (*textCellArray*) with the number of rows and columns the user has specified in the CellSizeForm. It then passes the *textCellArray* to its TextCanvas through the TextCanvas's *addBoard* method, so that the TextCanvas can draw the empty cells with the specified number in a row and a column. In this way, the CreateForm has direct access to the *textCellArray*, while the TextCanvas does not have direct access to the Board object (*board*).

This design supports the principle of low coupling of classes. Refreshing the TextCanvas is not necessary on the CreateForm, because each TextBox on the TextCanvas allows the user input or modification without refresh. In addition, the user may modify what he/she has entered in a cell later. Therefore, it is more efficient to send the information of all the cells when the user finishes specifying all the cells than to pass each user input to the *board* every time the user modifies the text in a TextBox.

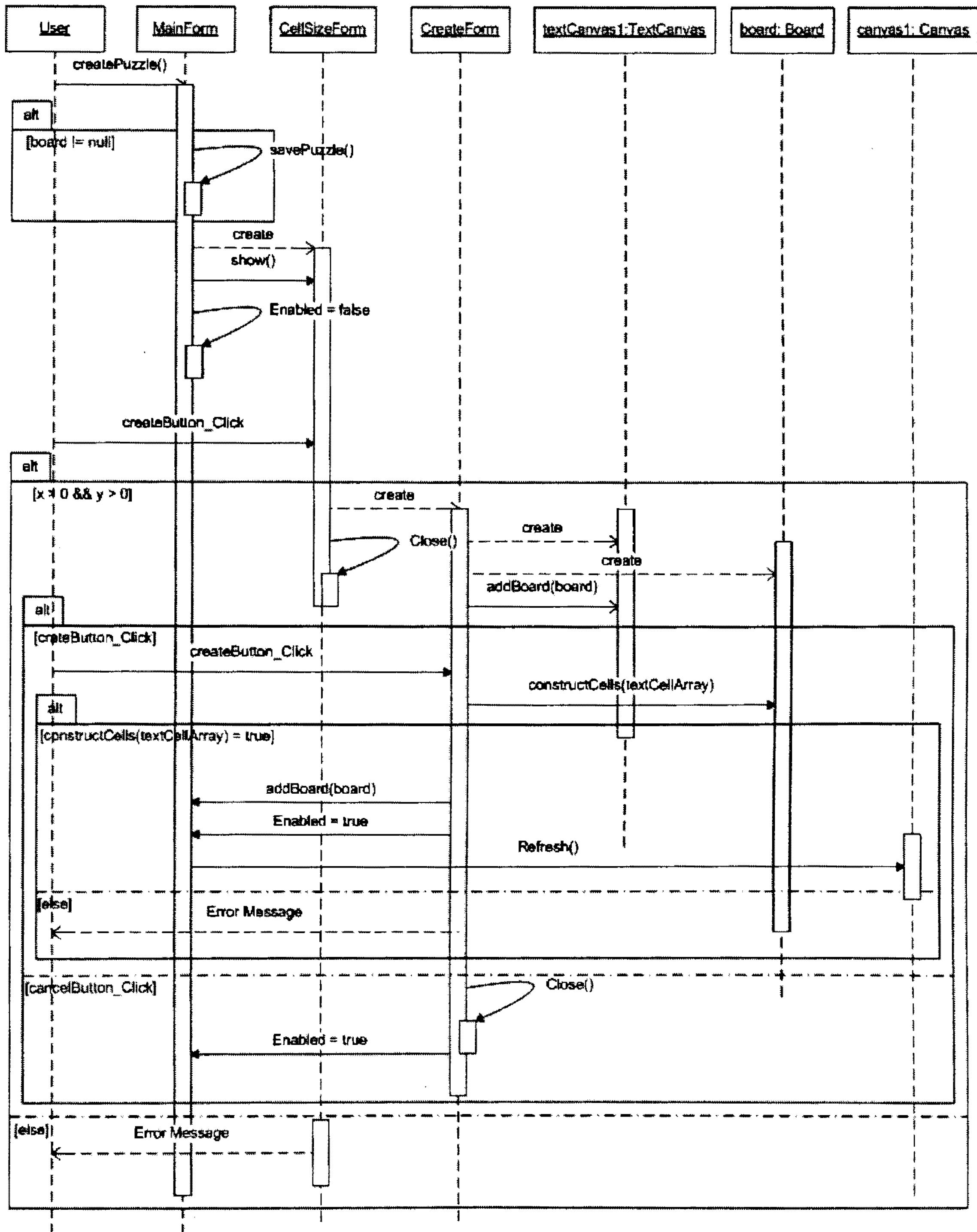


Figure 9. Sequence Diagram: Create a Puzzle

In each cell, the user enters the cell information such as clue number or a back-slash that represents the slash from top-left to bottom-right on black cells. Once the user

finishes specifying all the cells, the user should initiate the CreateButton_Click event. In the event procedure, the CreateForm creates the Board object *board*. It then calls the *board*'s *constructCells* method, passing the *textCellArray* as the parameter. The method then tries to initialize all the Cells in its two-dimensional Cell array called *cellArray*, and then returns whether all the Cells are initialized successfully.

If the *constructCells* method returns true, it indicates the user-created puzzle is valid in terms of the syntax of input, and the CreateForm passes the *board* to the MainForm through the MainForm's *addBoard* method. It then enables the MainForm, refreshes the MainForm, and closes itself. Refreshing the MainForm should also refresh the Canvas on it, so the puzzle should be displayed as the user specifies. If the *constructCells* method returns true, the error message should be shown as a MessageBox.

When the user opens a puzzle saved in a text file, the sequence is similar to the creation of a puzzle, though only the MainForm and the Board are involved. When the user calls the MainForm's *loadPuzzle* method, OpenFileDialog should be shown for the user to specify the file containing the puzzle. The MainForm then reads all lines in the file and passes the lines as a string array to the Board called *board*. The *board*'s *constructCells* method also takes a string array as the parameter, not just the two-dimensional TextBox array. If the method returns true, the MainForm refreshes its Canvas so that the puzzle should be displayed; if the method returns false, the error MessageBox should be shown.

3.2. Solve a Puzzle

Figure 10 is the sequence diagram for the process of completely solving a puzzle. When the user calls the *solvePuzzle* method by either clicking the Solve button or

selecting the Solve option on the menu, the MainForm first checks whether there is a puzzle currently loaded. If there is not, it displays an error message using a MessageBox; otherwise, the MainForm calls its *board*'s *solve* method. In the *solve* method, the Board attempts to find the answer by interacting with the Cells in its two-dimensional Cell array attribute called *cellArray*. The details of this method will be discussed in the Algorithm section.

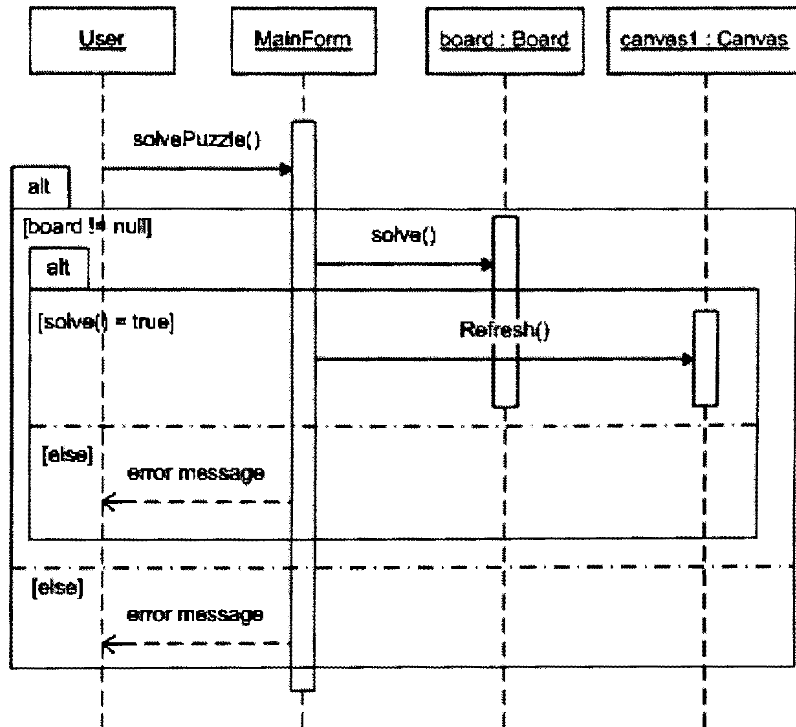


Figure 10. Sequence Diagram: Solve a Puzzle

The *solve* method returns true if the puzzle is completely solved. The MainForm then refreshes its Canvas object *canvas1* so that the answer should be displayed. If the method returns false, the MainForm shows an error message in a MessageBox, indicating that the puzzle cannot be solved.

3.3. Check User Inputs

Figure 11 is the sequence diagram for checking a puzzle to verify whether the user has entered numbers that are all correct and valid up to this point. The details of this task will be discussed in the Algorithm section. In this section the focus is on the interaction between the classes. When the user calls the *checkPuzzle* method by either clicking the Check button or selecting the Solve option on the menu, the MainForm first checks whether there is a puzzle currently loaded based on whether its Board object *board* is null. If the *board* is null, it displays an error message using a MessageBox; otherwise, it calls the *solve* method of another Board object called *answerBoard*. If the *solve* method returns false, the error message is displayed to notify the user that the puzzle cannot be solved. If it returns true, the MainForm passes the *answerBoard* to the *board* as the parameter for its *check* method.

The *check* method returns true if all the numbers entered by the user are same as the numbers in the corresponding InputCells of *answerBoard*. The MainForm then returns the message to let the user know that all the numbers entered so far are correct. If the *check* method returns false, MainForm returns an error message and refreshes the *canvas1*, so that the input cells with wrong numbers can be highlighted.

3.4.Enter a Number/Choice

In KakuroSolver, the user is allowed to click on the specific input cell on the Canvas and type a single numeric key (1-9) to enter the number in the input cell. To enter a number as one of the choices, the user should click on the input cell and press Shift key plus a numeric key. The same action (Shift + a numeric key) should be used to remove the number from the choices.

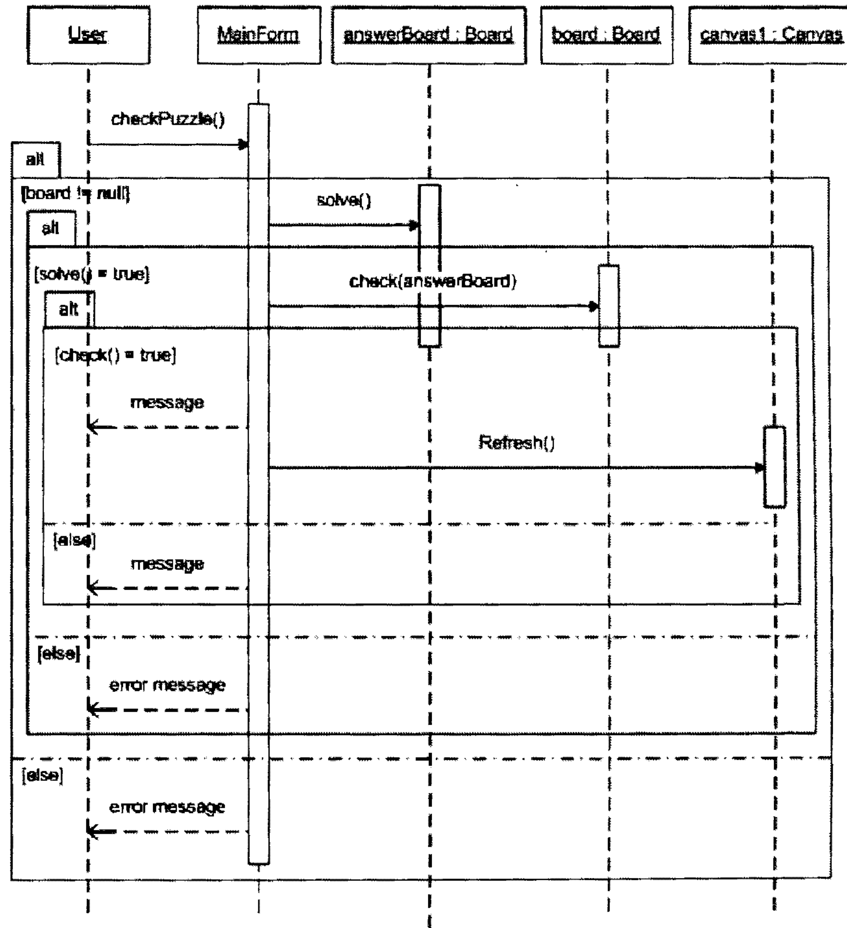


Figure 11. Sequence Diagram: Check a Puzzle

Figure 12 shows the sequence diagram for the *canvas1_KeyPress* event procedure of the MainForm. In C#, typing a single numeric key is recognized as a KeyPress event, which takes place "when the control has focus and the user presses and releases the key" [15]. This event, therefore, catches the user action to enter a number as a final in the puzzle.

When any key is pressed and released on the *canvas1*, the MainForm checks whether any input cell is currently selected, by looking at whether its *selectedX* and *selectedY* attributes have values larger than -1; these two attributes respectively represent the x and y locations of the selected input cell. If an input cell is selected, it is examined

whether the pressed key is a numeric key between 1 and 9. If it is numeric, the MainForm calls the *board*'s *getCell* method to get the reference to the corresponding InputCell. The user-entered number is then set to the InputCell through the InputCell's *setNumber* method, and the *canvas1* should be refreshed to display the entered number in the input cell.

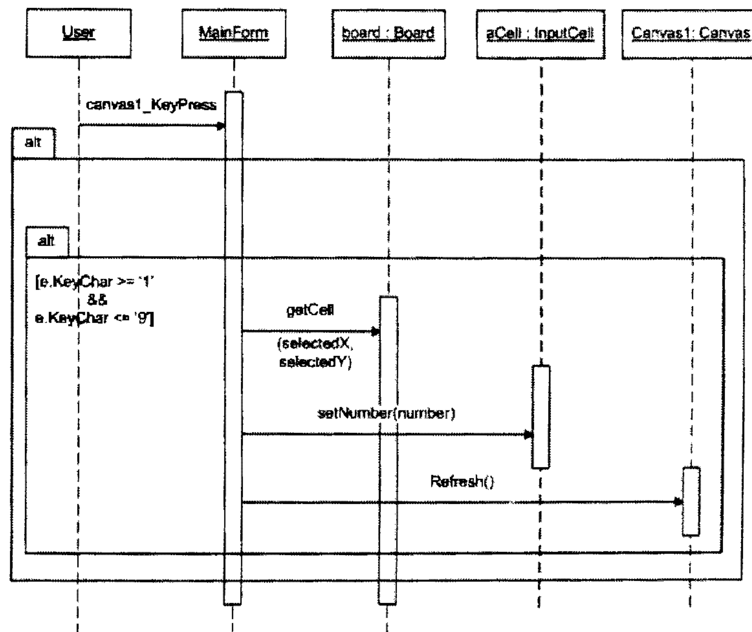


Figure 12. Sequence Diagram: KeyPress on Canvas

The user action to enter or remove a choice should be recognized as a KeyDown event on the *canvas1* (*canvas1_KeyDown*), which "occurs when the key is first pressed" [15]. This is a similar sequence as Figure 9. When any key is pressed on the *canvas1*, the MainForm checks whether any input cell is currently selected by looking at whether its *selectedX* and *selectedY* attributes have values larger than -1. If an input cell is selected, it is determined whether the Shift key is being pressed.

If the Shift key is pressed, the program should determine what other keys are also being pressed. If any numeric key between 1 and 9 is pressed along with the Shift key,

the MainForm calls the *board*'s *getCell* method to get the reference to the corresponding InputCell. The pressed number is then sent to the InputCell as the parameter of the cell's *setChoices* method, where the number is added to the InputCell's List<int> variable *choice* if the number is not contained in the *choice*. If the number is contained, it will be removed from the *choice*. After that the MainForm refreshes the *canvas1* to display the updated puzzle.

In addition, the input cell selection also follows the comparable sequence. The user action to select an input cell should be recognized as a MouseClick event (*canvas1_MouseClick*). When the user left-clicks somewhere on the *canvas1*, the MainForm determines the location of the mouse. The cell on which the user clicks with mouse should then be identified as follows.

selectedX = e.Location.X / *unitSize*

selectedY = e.Location.Y / *unitSize*

e.Location.X and Y respectively stands for the X and Y location of the mouse , and *unitSize* indicates the side of each square on the puzzle (grid). The MainForm then calls the *board*'s *getCell* method to get the reference to the corresponding Cell.

If the Cell is an InputCell, which is determined by the Cell's (and its children's) *getType* method, the MainForm sends the *selectedX* and *selectedY* to the *canvas1* as the parameters of the *canvas1*'s *setSelectedLocation* method. Finally, it refreshes the *canvas1* so that the selected input cell is framed with green color.

4. ALGORITHM

This section discusses several important functions of KakuroSolver and the algorithms that are used in each function.

4.1. Display a Puzzle on Canvas

In KakuroSolver, a puzzle is drawn on a Canvas object called *canvas1* in the MainForm, and the content of the puzzle, such as which type each cell is and which input cell has the number entered, is maintained by a Board object called *board*. The *canvas1* needs to be refreshed whenever the user makes a change to any input cell of the puzzle, so that the change is reflected in the puzzle and is visible to the user.

The *canvas1* has direct access to the *board*. For the first time the *board* is initialized, which means the puzzle is loaded, the *canvas1* decides the length of a side of the cells and the size of itself based upon the size of the puzzle. The *canvas1* then draws squares for each cell in the puzzle. For the input cells, the *canvas1* also draws additional information such as the numbers entered as finals or as choices.

The *canvas1* accesses each cell in the puzzle through the *board* to get the information for the cell. If the cell is a blank cell, a black square is drawn on the *canvas1*. If the cell is a clue cell, the *canvas1* draws a black square on which there is a white line from top-left to down-right. If the clue cell has a down clue, the down clue number is drawn in white on the right of the white line. If it has an across clue, the across clue number is drawn in white on the left of the white line.

If the cell is an input cell, a white square is drawn on the *canvas1*. In addition, the final number entered is drawn in black if the cell has the final number. If the input cell has some choices but no final number, those choices are drawn in black with smaller font

than that for final numbers. The input cell also maintains the validity of the final number, which indicates whether or not the number is correct. If the validity is false (invalid), the *canvas1* fills the square for the input cell with pink to let the user know the cell has an incorrect number. Moreover, the *canvas1* keeps the x location and y location of the input cell that is currently selected by the user. This location is changed whenever the user clicks the mouse on a different input cell on the *canvas1*. While the *canvas1* accesses the cell being selected, it draws a green frame on the cell, so that the user can know which cell he/she selects.

4.2. Create a Puzzle

When the user creates a new puzzle, all the cells need to be specified on the CreateForm. The CreateForm has a TextCanvas object called *textCanvas1* for the user to specify each cell in the puzzle. The *TextCanvas* class inherits the *Canvas* class, and thus it has direct access to the *board*, which maintains the entire puzzle. In addition to the variables that the Canvas has, the TextCanvas has a two-dimensional TextBox array with the same size as the *board*. Each TextBox in the array represents the corresponding cell in the *board*.

When the user specifies all the cells and clicks the "OK" button, the CreateForm sends the two-dimensional TextBox array to the *board* to initialize all the Cells in the *cellArray*. The *board* accesses each TextBox in the array, and takes what the user has entered in the TextBox. If the user has entered a back-slash in the TextBox, the *board* initializes the corresponding cell as a Cell object, which represents a blank cell. If the user has entered a back-slash with numbers before and/or after the back-slash, the corresponding cell is initialized as a ClueCell object, which represents a clue cell. The

number before the back-slash is recorded as the clue cell's down clue, and the number after the back-slash is recorded as the across clue. If the user has left the TextBox blank or entered one or more blank spaces, the *board* initializes the corresponding cell as an InputCell, which represents an input cell in the puzzle. As the user is creating a new puzzle, the input cell must not have any final number or choice.

When all the cells in the puzzle are initialized without any error, the *board* is passed to the MainForm. The CreateForm then refreshes the MainForm, which includes refreshing of the *canvas1* on the MainForm. The *canvas1* then accesses all the cells through the *board* in the way described in the "Display a puzzle on Canvas", and the puzzle is displayed as the user has specified. Finally the CreateForm closes itself and enables the MainForm.

4.3. Open a Puzzle

When the user wants to open a puzzle from a file, the MainForm shows the OpenFileDialog, which "prompts the user to open a file" [15]. The user then selects the file, and the MainForm checks whether the file specified by the user contains the puzzle in a correct syntax. First, the MainForm checks if the file does not exist or the file is empty, and if so, the error message is displayed.

If the file has contents, the MainForm reads the entire content of the file and sends it to the *board*. The *board* then initializes all the cells in the puzzle as specified in the file. The *board* reads one line at a time, which should represent one row of the puzzle.

In the file, each cell in a row should be split by vertical lines "|". If a back-slash is in the cell, the *board* initializes the corresponding cell as a Cell object, which represents a blank cell. If there is a back-slash with numbers before and/or after the back-slash, the

corresponding cell is initialized as a ClueCell object, which represents a clue cell. The number before the back-slash is recorded as the clue cell's down clue, and the number after the back-slash is recorded as the across clue.

Otherwise, the *board* initializes the corresponding cell as an InputCell, which represents an input cell in the puzzle. As the file may contain the partially solved puzzle, the input cell may have any final number or choice. If there is nothing or one or more blank spaces, the input cell is empty; there is no final number or choice entered. If there is a number between 1 and 9, the number is the final number for the input cell. If there is one or more numbers between 1 and 9 inside the parenthesis split by commas, those numbers are considered as choices for the input cell. The final number and the choices are both recorded in the input cell.

If all the cells are initialized without any error, the MainForm refreshes the *canvas1* so that the puzzle is displayed as specified in the file. Otherwise the error message is shown to let the user know that the file has one or more syntax errors.

4.4. Solve a Puzzle

The *solve* function tries to completely solve the puzzle and let the user know whether the puzzle is successfully solved or not. The solution should also be displayed in the puzzle. The *solve* method is in the Board class, and utilizes a backtracking search with several techniques that minimize the size of a search tree. As mentioned in 2.1.1.3, this approach can also be considered as a General Dijkstra's algorithm because the node with least candidates is always selected in each iteration.

Figure 13 shows the pseudocode of the *solve* method, which is recursive. In each recursion, the list of candidates for each empty input cell is updated with the forward

checking technique. The input cell with least candidates is then selected, and all the possible numbers (candidates) for the cell are entered as a trial one by one. If the rest of the puzzle can be solved with a possible number entered in the cell, the number should be the correct number for the cell.

```
Begin
  InputCell c = getBestCell() //get the input cell with least candidates among empty cells

  If c is not null Then
    List possibleNumbers = c.getPossibleNumbers() //get all the candidates for c
    For Each candidate in possibleNumbers
      c.setNumber(candidate)
      If solve() = true Then //recursion
        Return true
      End If
    End Loop

    c.setNumber(0)  //delete the last candidate entered in c
    Return false
  Else
    Return true
  End If
End
```

Figure 13. Pseudocode of the *solve* Method

To increase efficiency, each recursion employs the minimum remaining values heuristic and keeps the search tree as small as possible. It always selects the input cell that has the least possible numbers among empty input cells in the puzzle. In the *solve* method, all the cells in the puzzle are first visited, in order to find the input cell with the least possible numbers. The puzzle is represented by a two-dimensional Cell array called *cellArray*. As discussed in the Architecture section, the Cell class has two child classes: ClueCell and InputCell. These classes correspond to a clue cell and an input cell respectively. Therefore, the Cells represent the blank cells in the puzzle, which is not involved in the algorithm. The Cell class has the *getType* method that returns its type, and by calling this method, the type of each cell (Cell, ClueCell, or InputCell) can be identified.

As already mentioned, whenever an empty cell is filled with a number, the list of possible numbers for any other empty cell must be updated. To properly update all the possible numbers lists, all the cells are visited from left to right, then from top to bottom.

It is because when an input cell belongs to a down run, the clue cell that contains the clue for the down run should be in the same column as the input cell, but on the upper side of it. Similarly, if an input cell belongs to an across run, the clue cell that contains the clue for the across run should be in the same row, but on its left. Therefore, when the cells are visited in this order, an input cell is always visited after its down clue cell and/or its across clue cell. Moreover, for any input cell that has both the down clue and the across clue, the down clue cell for the input cell is visited before the across clue cell.

The list of possible numbers for each input cell is assigned by looking at clue cells. Each clue cell maintains how many input cells are associated with its down clue and across clue. When a clue cell with a down clue is visited, the possible numbers for the empty input cells that belong to the clue cell's down run are identified and set to those empty input cells. When the clue cell that has an across clue is visited, the possible numbers for the empty input cells that belong to the clue cell's across run are identified and set to those empty input cells.

In the example shown in Figure 14 below, the cell A1 is visited first. The next cell B1 is a clue cell with the down clue of 21, which has three input cells associated with it. Here the possible numbers for the run with clue of 21 and three input cells are identified, and these numbers are added to the lists of possible numbers of the input cells B2, B3, and B4. The cells C1 and A2 are then visited, and the next is the input cell B2. As B2 does not have an across clue, the possible numbers identified in B1 are all possibly

entered in it at this point. After that the clue cell C2 is visited, the possible numbers for the input cells C3 and C4 are identified. In the next input cell A3, the possible numbers for B3 and C3 are discovered. However, the possible numbers for B3 are already set when B1 is visited, and those for C3 are also set when C2 is visited. For B3, therefore, the possible numbers found in B1 and those found in A3 are compared. Only the numbers common to both lists of possible numbers should be left as the possible numbers for B3. The possible numbers for C3 are also determined in the same way. The same step is taken for A4, B4, and C4.

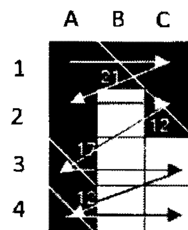


Figure 14. Order in which Cells are Visited

The possible numbers for a run are identified as follows. First, all the input cells that belong to the clue cell's down or across run are visited to see if any of these cells already contain the final number. Any numbers that are already entered cannot be entered in the empty input cells in the same run, and so these numbers are recorded in the invalid numbers list. Any number in this list cannot be entered in an empty cell of this run.

Next, the maximum possible number and the minimum possible number for the run are identified, based on the number of empty input cells and the sum of numbers that should be entered in these cells. The sum may be different from the down clue, when one or more input cells in the run already contain numbers.

The algorithm used to identify the maximum possible number utilizes the largest/smallest possible numbers technique discussed in 1.2.3. When the number of cells

is 1, the value of clue should be the maximum possible number; if it is n that is greater than 1, the maximum possible number should be calculated by (clue – (1 + 2 + … + (n – 1))). The minimum possible number should be calculated in the same way. When the number of cells is 1, the value of clue should be the minimum possible number; if it is n that is greater than 1, the minimum possible number should be calculated by (clue – (9 + 8 + … + (10 – (n – 1)))).

The maximum or minimum possible number identified may not be between 1 and 9. If the maximum possible number is greater than 9, it means that the maximum possible number is 9; this indicates there is no number that can be added to the list of invalid numbers for the run. When the maximum possible number is less than 9, all numbers that are greater than the maximum possible number and less than or equals to 9 should be added to the list of invalid numbers. Similarly, if the minimum possible number is less than 1, the minimum possible number for the run should be 1, and no number can be added to the list of invalid numbers. When the minimum possible number is greater than 1, all numbers that are less than the minimum possible number and greater than or equals to 1 should be added to the list of invalid numbers.

At this point, all the numbers that cannot be entered in the empty input cells should have been identified and contained in the list of invalid numbers. Therefore, the numbers in the list of invalid numbers are removed from the list of possible numbers for any of the empty input cell when they are contained in the list. By default, the list of possible numbers contains the integers between 1 and 9. Thus the remaining items in the list of possible numbers should be the numbers that may possibly be entered in the input cells.

At the time when an empty input cell is visited, the possible numbers for the cell should have been already identified. This is because both the clue cells for the down run and the across run that it belongs to are already visited. The default possible numbers for the input cell are 1-9, but the invalid numbers identified should have been removed from the list of possible numbers. All the numbers that are still in the list of possible numbers may be entered in the input cell at this point.

Therefore, when an empty input cell is visited, the possible numbers for the cell are counted. If the count is 0, it indicates that the number that has previously been entered in another input cell must be wrong. The current recursion must then return false. This indicates that the number entered for trial in the previous recursion is incorrect.

If the count is exactly one, this cell must have the least possible numbers. The current step of finding the input cell with the least possible numbers is terminated, and the process goes to the next step.

If the count is more than one, it is compared with the least possible numbers to the point. If the count is smaller than the current least possible number, the current input cell should be recorded as the input cell with the least possible number. Unless there is any input cell that has zero or one possible number, all the cells in the puzzle should be visited, and the input cell with the least possible numbers should be identified.

Once the current recursion determines which input cell it should work on, the list of possible numbers for the input cell should be looked at, and the first item (integer) in the list should be entered as the final number for the cell, assuming that the number is correct. Another *solve* method is then called as a next recursion to see whether the rest of the puzzle can be successfully solved with the number.

If the *solve* method in the next recursion returns true, it means that the rest of the puzzles should be able to be solved with the number entered. The number should be correct, and therefore the current *solve* returns true with the number entered. If the *solve* in the next recursion returns false, the number is incorrect for the cell, and the next item in the list of possible numbers should be entered. When all the items in the list are turned to be incorrect, or when there becomes at least one input cell that has no possible number, the current *solve* should return false after it clears the last number entered as a trial in the input cell.

When the entire puzzle is solved, the canvas is refreshed. As numbers should have been identified for all input cells, all input cells in the puzzle should be displayed with the numbers filled.

4.5. Check User Inputs

The *check* method determines whether the user's inputs to the point are all correct. When a puzzle is created through the application or opened from a text file, the puzzle is created, which is represented as a Board object called *board*. This puzzle should interact with the user and stores the user inputs. Additionally, the clone or copy of the puzzle called *answerBoard* is created in the MainForm.

When the user initiates the *check* method, the puzzle stored in the *answerBoard* is solved by the application using the *solve* method. If the *answerBoard* is solved successfully, all the input cells on the *answerBoard* should be filled. The number entered in each input cell on the *board* is then compared with the number identified in the corresponding input cell on the *answerBoard*. If the application cannot solve the *answerBoard*, the user will be notified through an error message.

If the *answerBoard* is solved but an input cell on the *board* has the number that does not match that of the *answerBoard*, the input cell is considered to have an invalid number. Each input cell maintains the status of whether the number entered in it is correct, and so the status will be set to invalid (false). If an input cell has the same number as the corresponding input cell on the *answerBoard*, the status of the validity of the number entered will be valid (true).

After all the input cells on the *board* are compared with the corresponding input cells on the *answerBoard*, the canvas will be refreshed. The canvas checks the validity of the number for each input cell whenever it is refreshed, and so those input cells with invalid numbers are highlighted with pink color. Then the user is able to know the numbers entered in those cells are wrong. In addition, the user will receive the message of whether there is any incorrect input or not.

4.6. Select an Input Cell

When the user clicks the mouse on the *canvas1* to select an input cell, the MainForm recognizes the event. It then determines the location of the mouse, and identifies on which cell the mouse is clicked. If the cell is not an input cell, the event is disregarded. However, if the cell is an input cell, the MainForm passes the location of the cell to the *canvas1* and refreshes the *canvas1*. Subsequently the *canvas1* redraws itself with the change of the selected cell, using the algorithm described in the "Display the puzzle on Canvas".

4.7. Enter/Delete a Number

When the user presses a numeric key to enter the number as a final in an input cell, the MainForm recognizes the event. It then checks which cell is currently selected. If

any input cell is selected, the MainForm determines which key is pressed. If the key pressed is not the numeric key between 1 and 9, the event is not distinguished as entering a number. It is disregarded unless the key pressed is related to any other function such as entering a choice and using a shortcut. If it is the numeric key 1-9, the MainForm passes the number entered to the *board* so that the *board* set the number to the input cell being selected. After that the MainForm refreshes the *canvas1*. Since the *canvas1* accesses all the cells through the *board* whenever it is refreshed, the number newly entered in the input cell is drawn on the cell.

The same process is required to delete the number from an input cell. The only difference is the key pressed by the user. The user needs to press the Delete key to delete the number entered in an input cell. The MainForm recognizes this event, and checks which cell is currently selected. If any input cell is selected, the MainForm determines which key is pressed to find that it is the Delete key. The MainForm then passes the number 0 to the *board* so that the *board* set the 0 to the input cell being selected. The input cell is considered to have no number entered, when 0 is set as its number. After that the MainForm refreshes the *canvas1*. Since the *canvas1* accesses all the cells through the *board* whenever it is refreshed, the input cell is drawn as an empty cell.

4.8. Enter/Delete a Choice

When the user presses the Shift key along with a numeric key to enter the number as a choice in an input cell, the MainForm recognizes the event. The exact same event is necessary to remove a choice from the input cell.

When the MainForm notices this event, it checks which cell is currently selected. If any input cell is selected, the MainForm determines the key(s) currently pressed. If the

key pressed is not the numeric key between 1 and 9, the event is disregarded unless the key(s) pressed are involved in any other function. If the numeric key 1-9 is pressed, the MainForm passes the number entered to the *board* as a choice.

The *board* then examines whether the choice passed in exists in the list of choices for the input cell being selected. If it does not exist, the *board* will add the number to the list of choices for the input cell. If it exists, the number will be removed from the list of choices for the input cell. After that the MainForm refreshes the *canvas1*. Since the *canvas1* accesses all the cells through the *board* whenever it is refreshed, the input cell is drawn with the updated list of choices.

4.9. Clear All Inputs

The *clear* function clears all the user inputs, which are the final numbers and the choices entered in the input cells. When this function is triggered, the MainForm calls the *clear* method of the *board*. The *board* then accesses all the input cells in the puzzle, which are stored as InputCell objects in the two-dimensional Cell array.

As already described, each input cell maintains the final number and the list of choices for itself. For each input cell, the *board* sets the number 0 as its final number and deletes all the numbers in the list of choices. After all the input cell is updated by the *board*, the MainForm refreshes the *canvas1*. Since the *canvas1* accesses all the cells through the *board* whenever it is refreshed, all the input cells are drawn without any final number or choice.

4.10. Save a Puzzle

At any time when the puzzle is displayed, the user can save the puzzle with final numbers and choices that are currently entered up to this point. In addition, the puzzle

may also be empty or completely solved. When the user triggers this function, the MainForm shows the SaveFileDialog, which "prompts the user to select a location for saving a file" [11]. When the user selects an existing file, the SaveFileDialog asks whether the user wants to overwrite the file. If the user does not want to overwrite the file, the user has to specify a new file for saving.

The MainForm then accesses all the cells in the puzzle through the *board* from left to right and top to bottom. Each cell has the *toString* method that returns all the information for the cell as a string. The MainForm calls the method for each cell, and write the string returned in the file. When the MainForm moves from one cell to the next cell, it writes a vertical line "|" so that the cells are separated. When one row is completed, the MainForms writes the cells in the next row in the next line of the file. The file then contains one row in one line.

5. EVALUATION

This section evaluates KakuroSolver and discusses the possible enhancements that the application may have in the future.

5.1. Evaluation of KakuroSolver

The application has been developed following the spiral model, which is one of the evolutionary software process models. This model "couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model" [16]. First, the paper model was produced in a cyclical process, focusing on how the application should behave for different user actions. The actual application was then designed with details and developed in a sequence of evolutionary cycles.

For example, the MainForm was first developed. All the controls were placed according to the paper model, and the Canvas class was implemented so that it can adequately draw an entire puzzle. The function to open a puzzle was then implemented, which enabled testing of the Canvas with multiple puzzles.

Whenever a method was implemented in a class, the method was tested in isolation with several white-box and black-box unit tests before it was actually called by another method or class. This made it easier to identify where the error was in the code, and it decreases the errors that might be found in the later iterations.

The code was written by utilizing the principle of design by contract, in which classes characterize their behavior and interaction by contracts. A contract "consists essentially of a class invariant, pre- and post-conditions for all the methods of the class interface" [14]. The idea of design by contract was introduced to the classes that would not directly interact with the user. It was because predicting all user behaviors was

impossible. However, employing the design by contract in the back-end classes reduced the number of conditions to check whether the client (caller) satisfies the preconditions of a method, and the performance of the classes and methods were improved.

The code also focused on the readability. A method with long lines of code was divided into some smaller methods, and the duplication of code was minimized by making the duplicated piece of code an independent method. Also classes, methods, and variables were named so that they indicated their purposes. In addition, the comments were written for each method in a class to specify what the method should do.

Once the black-box testing was complete and all the classes were integrated, the application was tested with system testing. This application aims at solving Kakuro puzzles with any size and difficulty. Therefore, based on these criteria below, 50 Kakuro puzzles were selected for testing of the application from numerous online sources:

- When the puzzles are categorized by difficulty, select the ones with different sizes but highest difficulty
- When the puzzles are categorized only by size, select the largest ones

These criteria are determined based upon the fact and assumption that the puzzles become more difficult when they become larger, and that the application should be able to solve the easy puzzles when it can solve difficult ones.

As part of the testing, all 50 puzzles were first created through the application and solved. The same puzzles were also created as text files, opened in the application, and solved. As a result, 50 out of 50 puzzles were loaded and solved successfully. Here the success means that the application was able to fill all the input cells in a puzzle following the rule of Kakuro listed below:

- Each input cell may contain the number between 1 and 9
- The same number cannot be entered in the same run, though the same number may be entered in different runs on the same row or column
- The sum of numbers in a down run should equal the down clue for the run; the sum of numbers in an across run should equal the across clue for the run

There were times when the puzzle solved by KakuroSolver was slightly different from the solution provided by the source. However, it was all because there were multiple possible solutions. An example is shown in Figure 15. This puzzle was provided by kakuropuzzle.com [10], and the left puzzle is the solution attached to it. The right puzzle is the solution solved by KakuroSolver. Both solutions in fact satisfy the rules above, and hence considered to be correct. The well-formed Kakuro puzzles should have only one possible solution, and so this inconsistency may not be a serious issue.

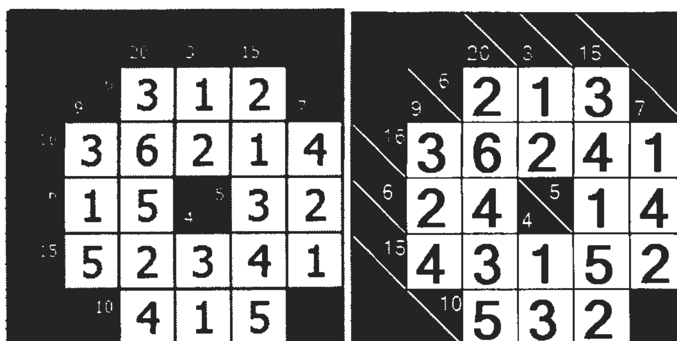


Figure 15. Kakuro Puzzle with Multiple Solutions

In terms of the time for the application to solve a puzzle, it was less than a second to solve any puzzle regardless of its size. This time taken should be satisfactory not to annoy the user. Therefore, it is concluded that the *solve* function of the application should have fairly good performance in terms of both time and accuracy.

For the algorithm used in the *solve* function, a search algorithm must be employed in order to find the solution for any Kakuro puzzle if there is one. Using only logical reasoning may not find a solution for a puzzle if it contains unexpected components. As for the searches, generally as the search tree becomes more complex, other search algorithms mentioned in 2.1.1 will be more efficient than a backtracking search. As a trade-off, however, those algorithms are more difficult and complicated to implement. Considering the limited development time, it was decided to use a backtracking search. Instead, the backtracking is together with all the techniques mentioned in 2.1.1.1. To identify candidates for an input cell, the largest/smallest possible numbers technique is utilized to reduce the number of candidates. In each iteration of the function, the empty cell with least candidates is chosen, so that the search can be minimal. The list of candidates for each empty input cell is updated whenever an empty cell is filled with a number. The number of candidates for each cell is, therefore, reduced as more input cells are filled. With these techniques, the search is minimized, and the backtracking search is fast enough to find the solution for even a difficult puzzle.

Based on the solution generated by the application, the *check* function was also tested. The *check* function was called after about one-third of the input cells were filled with the same numbers as the solution, and another one-third were filled with different numbers. The result was for all 50 puzzles, every number that was different from the solution was highlighted as an invalid number. This also took less than a second from initiation (clicking the Check button or selecting the Check option from the menu) to completion. This function also worked acceptably in terms of time taken.

In terms of accuracy, however, there is still room for improvement. As already discussed, the *check* function currently checks the user inputs only against the solution generated by the application. The application provides fairly accurate judgment about whether an input cell in the puzzle has the same number as the corresponding input cell in the solution. However, if the puzzle has multiple possible solutions, and if the user enters the numbers so that they match with another solution but do not match with the solution generated by the application, the user receives the message saying that there are incorrect numbers, although the numbers are actually correct. For example, when 3 is entered in the top-left input cell in the puzzle shown in Figure 15, it is considered to be incorrect, though it is the same number as the solution provided by the source (see Figure 16).

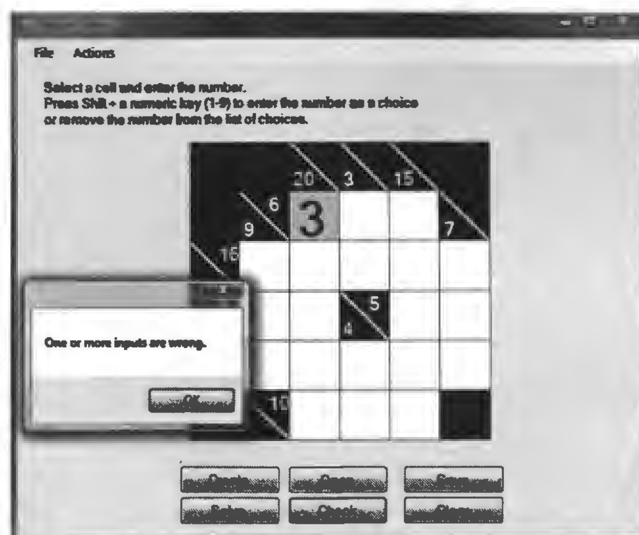


Figure 16. Screenshot: *check* Function Returning Incorrect Result

On the function to create a puzzle through the application, the performance was also acceptable. In general the size of large Kakuro puzzles is at most 20 x 20. It took about a second after clicking the Create button on the CellSizeForm to display the 20 x 20 puzzle on the CreateForm. The smaller puzzles took less than a second.

Although KakuroSolver is presentable overall, there are some known issues. One of them is that the application requires the .Net Framework 3.5 installed and updated on the machine. Otherwise the user may receive the error message displayed in Figure 17 when the user clicks the Create button on the CreateForm.

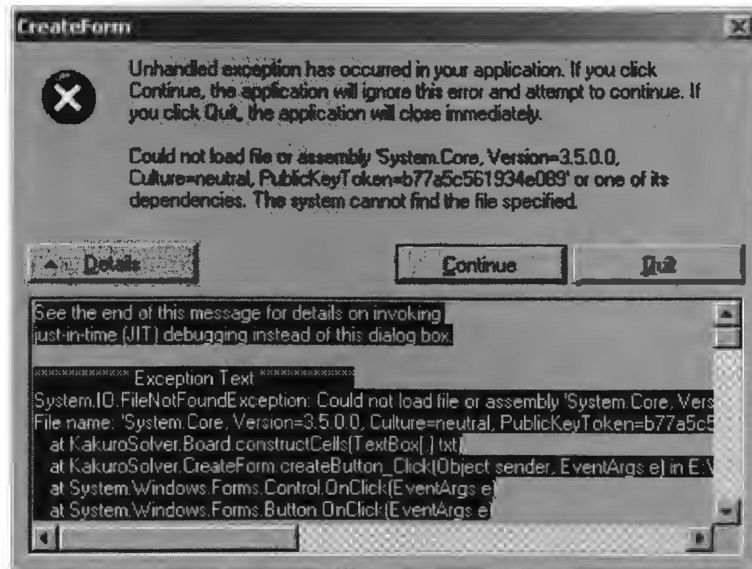


Figure 17. Screenshot: Error Message regarding .Net Framework 3.5

Another known issue is that the puzzle blinks whenever the user makes changes to the puzzle, such as entering a number and deleting a number. This is because the Canvas needs to be refreshed in order to reflect the change to the puzzle and make it visible to the user.

5.2. Future Enhancements

The most significant future enhancement is to make KakuroSolver handle the puzzles with multiple possible solutions. Currently when the user checks whether the user input to the point is correct, the application looks at only one solution generated by its *solve* function. The user then may receive the message that one or more numbers are

incorrect, though they are all correct when it is checked against the other (correct) solution of the puzzle.

The testing shows that a smaller puzzle is more likely to have multiple solutions than a larger and more complicated puzzle. Considering the fact that a novice player starts with smaller and easier puzzles, the user may be confused with the message the application displays in the *check* function.

One possible solution for this is to make the application try all the possible numbers in the *solve* function. At this time, the *solve* function ends when a solution is found. Theoretically the *solve* function should be able to identify all the possible solutions if it keeps trying the possible numbers even after a solution is found.

To implement this idea, the *solve* method in the Board class needs to be modified. The existing *solve* method returns a boolean value, which is either true or false, indicating whether the puzzle is completely solved. Instead, it may return a list of two-dimensional Cell arrays, where each array in the list contains a single possible solution. Then if the list returned contains at least one item, it is said that the puzzle can be solved; if there is no item in the list, it means that the puzzle cannot be solved.

However, this change may cause another issue. In case there are multiple possible solutions for a puzzle, one solution has to be selected against which the user inputs are checked. When the puzzle is 90% solved, it may be possible to identify which solution is the closest to the user inputs so far. Yet most of the time, the user inputs may partially match with a solution, but may partially match with another solution as well. Then the application may sacrifice highlighting the input cells with wrong numbers and simply display the message indicating whether there is any incorrect number entered. In this way,

the *check* method checks the user inputs against all the possible solutions, and returns whether or not there is any solution that match with the user inputs to the point.

Another possible enhancement is to allow undo and redo. At present, the user himself/herself has to reverse a user action. For example, if the user enters a choice in an input cell by mistake, he/she has to select the cell and press the Shift key along with the numeric key to remove the choice from the list of choices. If the user enters the number as a final instead of a choice, the user needs to press the Delete key to delete the number entered, and then press the Shift key together with the numeric key to enter the number as a choice. It is not very time-consuming, but allowing the user to undo and redo the actions will likely increase the user satisfaction and the application's usability.

To implement the undo and redo, two queue structures need to be added: One for undo and the other for redo. Whenever the user makes changes to the puzzle, the action should be recorded in the undo queue. If the user decides to undo several actions, those actions reversed should be recorded in the redo queue. The user then should be able to redo when the application reads the redo queue and reverse those actions.

The function to allow the user to modify the puzzle may also be added to the application. There are times when the user creates the puzzle through the application, and later notices that one of the clue numbers is wrong. Presently the only way to fix the puzzle is to save the puzzle in a text file, and modify the puzzle there. Allowing the user to modify the structure of the puzzle through the application will be helpful to the user.

Another approach to address this issue is to check whether the puzzle can be solved before the CreateForm is closed. The application then notifies the user when the puzzle being created cannot be solved, and forces the user to review what the user has

entered in the CreateForm. This approach is dangerous when there may be puzzles that the application cannot solve but has a solution. However, this approach may be taken after exhaustive user testing.

In addition, the function that allows the user to move from one cell to another by using the arrow keys may be added to the application. Actually this function was tried to be implemented, but pressing an arrow key on the Canvas was not caught as the KeyDown event properly. When the user can use arrow keys to move between cells, the user movement can be minimized. The user does not have to click a cell, type in a number, and go back to the mouse to click another cell. Instead, the user can use the keys on the keyboard for all the inputs, once the user clicks the mouse to select the first cell.

Another function that may be added to the application is to show the hint to the user. For example, when the user clicks the Hint button, a couple of empty input cells are filled with the correct numbers and then becomes empty after some time. This function may not go well with the new *check* function discussed earlier in this section, but should work fine with the current *check* function that has only one solution available.

6. USER MANUAL

This section provides the user manual and is intended to guide the user of KakuroSolver with the entire functionality and features.

As an overview, this application solves Kakuro puzzles of any size and difficulty. It has two modes: One in which it reads and completely solves the puzzle, and the other in which it reads the puzzle and allows the user to solve the puzzle while maintaining lists of possible choices for the input cells in the puzzle.

The Figure 18 is the initial screen of the application, which is the main window of the application. All options are available, both in the menu and as the buttons. Several options are disabled until a puzzle is loaded to the screen. The rest of this section describes each option of the application.

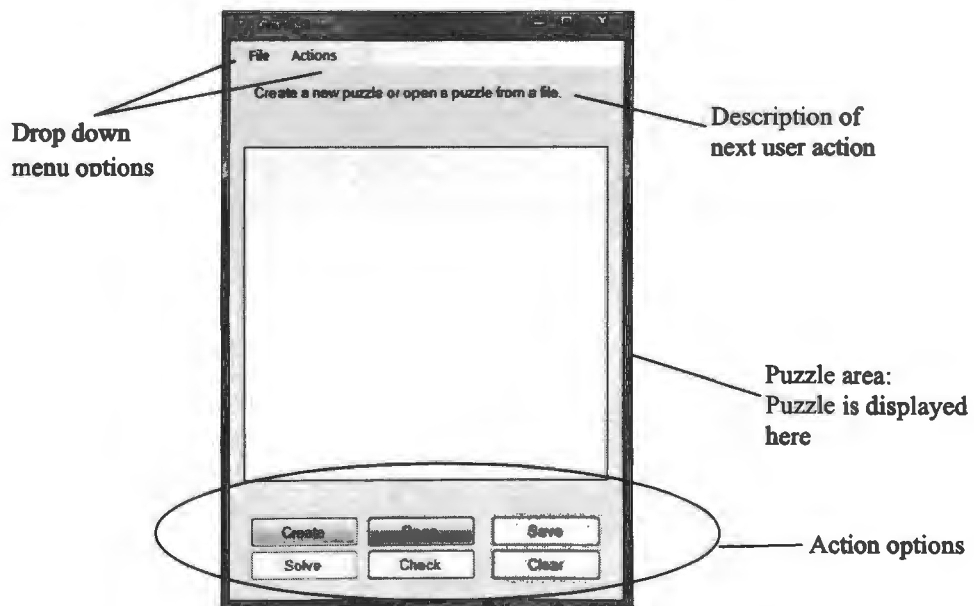


Figure 18. Screenshot: Initial Screen (Main Window)

6.1. Create a Puzzle

The user can create a new puzzle in two ways: Create it through the application, or create it as a text file and open it.

6.1.1. Create a puzzle through the application

When the user creates a new puzzle through the application, the user can do one of the following:

1) Click the Create button
2) Go to the menu bar and select File → Create New Puzzle
3) Press Ctrl + N

When there is already a puzzle loaded on the screen, the message "Do you want to save the current puzzle?" will be displayed. The user will then need to click "Yes" to proceed after saving the current puzzle, click "No" to proceed without saving the current puzzle, or click "Cancel" to go back to the main window. If no puzzle has been loaded, the user can directly go to the next step.

In the next step, the pop-up window appears for the user to specify the size of the puzzle (see Figure 19). The user can click "OK" after entering the width and height of the puzzle, or click "Cancel" to go back to the main window. The width and height must be both positive integers.

If the user clicks "OK" with valid inputs (width and height), the new window shown in Figure 20 opens where the user specifies each cell of the puzzle. On the left of the screen, the user may find a brief instruction of how to specify each cell. If the user wishes to specify an input cell, the cell should be left blank. If the user wishes to specify a black blank cell, a back-slash should be entered in the cell. If the user wishes to specify a clue cell, the user may type the down clue number followed by a back-slash and by the across clue number. If the user wishes to specify a clue cell that has either the down clue

or across clue, but not both, the user may leave it blank for the unused clue. For example, the input cell only with the across clue 12 should be entered as "\12".

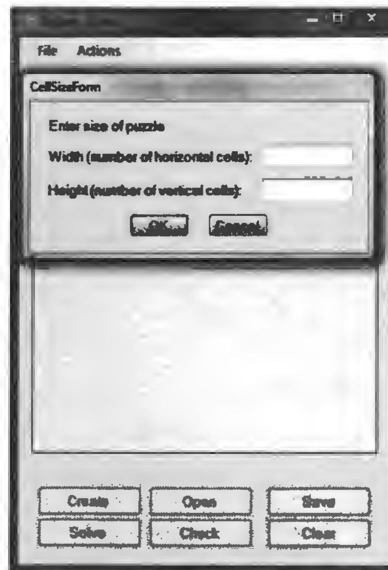


Figure 19. Screenshot: Specify the Size of Puzzle

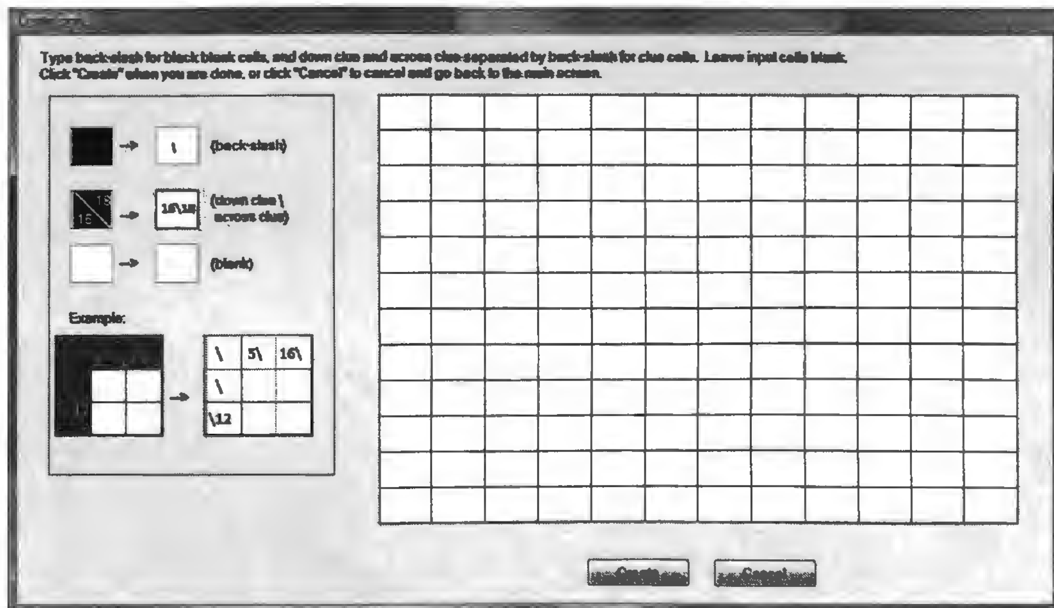


Figure 20. Screenshot: Create a Puzzle

Once all the cells are specified, the user can click "Create" to load the puzzle. When the user's inputs follow the syntax, the puzzle should be created. The user can then go back to the main window, which now displays the puzzle created in the puzzle area. If

any of the cells contain the input against the syntax (blank or a back-slash with one or two numbers), the error message should be shown when the user clicks "OK" button. At any time of this process, the user can click "Cancel" to quit creating the puzzle and go back to the main window.

6.1.2. Create a puzzle in a text file format

When the user creates a puzzle as a file, the file must be a text file and follow the syntax that is shown in Table 2 below. Each line should represent a single row, with vertical line (|) splitting each cell in a row. The syntax for each cell is the same as when we create a puzzle through the application. A blank space or no space represents an input cell, and a back-slash represents a blank cell. A clue cell is the down clue number followed by a back-slash and by the across clue number. If a clue cell only has one clue, the other clue can be blank or no space.

Table 2. Syntax of Puzzle in a Text File Format

Type of cell	Syntax
Input cell	blank space or no space
Blank cell	\
Clue cell	down clue number \ across clue number (i.e., 12\16)

Figure 21 is the example of a text file; the size of this puzzle is 8 x 8. When the user opens the puzzle in Figure 21, the puzzle is displayed in the puzzle area of the main window (see Figure 22 for screenshot). See 6.2 below for the explanation of how to open the puzzle created in the application.

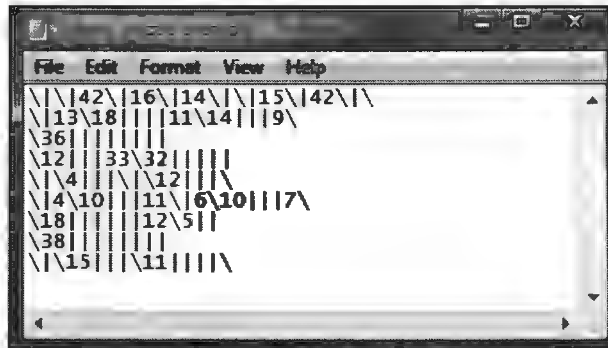
6.2.Open a Puzzle from a File

The user can open a puzzle from a text file. There are several ways to do it:

1) Click the Open button,

2) Go to the menu bar and select File → Open Puzzle from File, or

3) Press Ctrl + O



```
\|\|42\|16\|14\|\|15\|42\|\
\|13\18||||11\14|||9\
\36||||||||
\12|||33\32|||||
\|\4|||\|\12|||\
\|4\10|||11\|6\10|||7\
\18||||||12\5||
\38||||||||
\|\15|||\11||||\
```

Figure 21. Example of Puzzle in a Text File Format

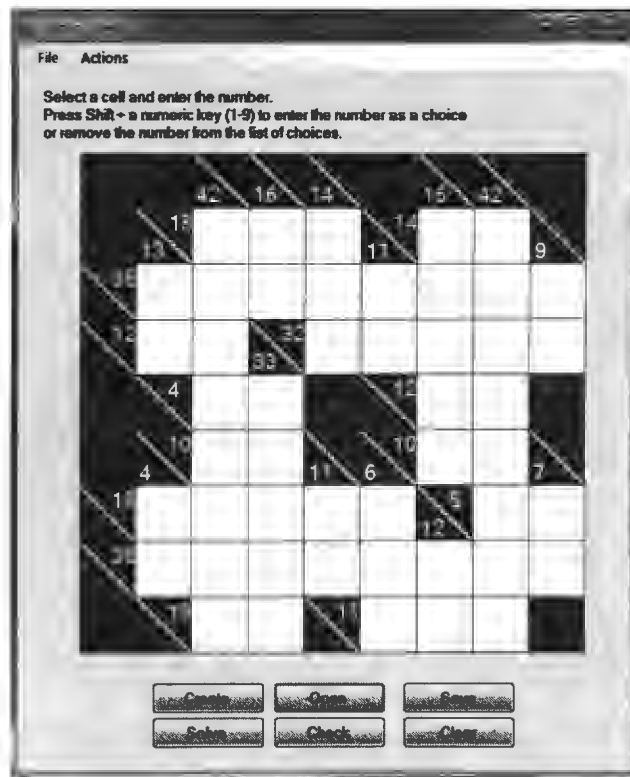


Figure 22. Screenshot: Example of Puzzle Displayed in the Main Window

When there is already a puzzle loaded on the screen, the message “Do you want to save the current puzzle?” should be displayed. The user needs to click “Yes” to proceed after saving the current puzzle, click “No” to proceed without saving the current puzzle,

or click "Cancel" to go back to the main window. If no puzzle has been loaded, the user can directly go to the next step.

In the next step, the new window (shown in Figure 23) pops up to ask the user to specify the file that contains the puzzle. The user needs to click "OK" after selecting the file, or click "Cancel" to go back to the main window.

Figure 23. Screenshot: Open a File containing a Puzzle

If the user-selected file is in correct syntax, the puzzle should be displayed in the puzzle area on the main window; if not, the error message should be shown. If partially solved puzzle is opened, the numbers and choices should be also displayed in the puzzle. The user should receive an error message if the user specifies a puzzle that does not exist, is an empty file, or has any syntax error.

6.3. Solve a Puzzle

After a puzzle is displayed in the puzzle area on the main window, the user may use the application to completely solve the puzzle in two ways:

1) Click the Solve button
2) Go to the menu bar and select Action → Solve Puzzle

When the application can solve the puzzle without any error, the answers should be displayed in the puzzle area. Figure 24 is the screenshot when the application solves the puzzle shown in Figure 22. When the application cannot solve the application, the user should receive an error message "The puzzle cannot be solved".

Figure 24. Screenshot: Puzzle Solved by KakuroSolver

The user can use this *solve* function after entering some numbers in the input cells; however, the user may receive the error message "The puzzle cannot be solved" when the user enters one or more numbers that are incorrect.

6.4. Enter a Number in an Input Cell

To enter a number in an input cell, the user first needs to select the cell by left-clicking the mouse anywhere on the cell. The selected cell should then be framed in green. In the example shown in Figure 25, the upper-left input cell on the first row is selected. Once the input cell is selected, the user can type the number (1-9); the entered number should then be displayed in the selected cell (see Figure 26).

6.5. Delete a Number from an Input Cell

When the number entered in an input cell turns out to be incorrect, the user can delete the number, by pressing the Delete key after selecting the cell.

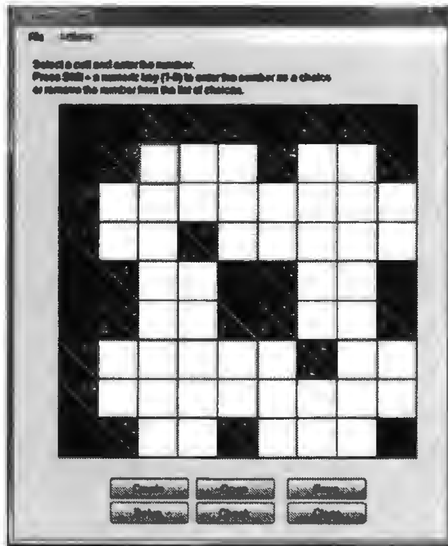


Figure 25. Screenshot: Select an Input Cell in the Puzzle

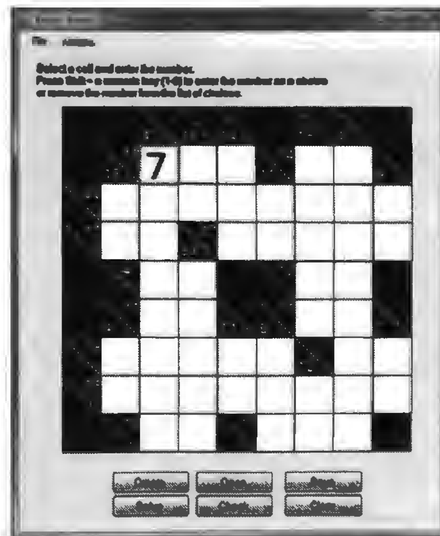


Figure 26. Screenshot: Enter a Number in the Selected Input Cell

6.6. Enter a Choice in an Input Cell

When the user assumes there are multiple numbers that may potentially be entered in an input cell, the user can enter these numbers as a choice in the cell. To do this, the

user first needs to click the input cell in which he/she wants to enter a choice in the way described in 6.4. The user should then press and hold the Shift key and the numeric key (1-9) of choice. Figure 27 shows when four choices (1, 2, 4, and 7) are entered in an input cell. An input cell can have up to nine non-overlapping numbers as choices.

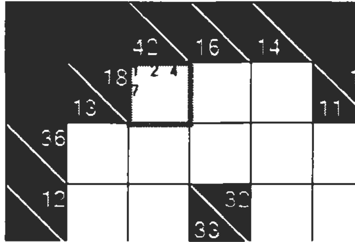


Figure 27. Screenshot: Enter a Choice in the Selected Cell

When the user decides which number should be entered as the final number for an input cell, the user selects the cell and type the number without pressing the Shift key, as is explained in 6.4. The final number should then be entered in the cell, overwriting all currently existing choices. The application saves the choices even after the choices are overwritten by the final number. Therefore, if the user deletes the final number later for whatever reason, the choices should reappear in the cell again.

6.7. Remove a Choice from an Input Cell

The user can delete a choice in the same way of entering the choice by pressing the Shift key and the numeric key. For example, if the user presses the Shift key and "4" in the puzzle shown in Figure 28, the choice 4 should be removed from the list of choices and only 1, 2, and 7 are displayed as choices for the cell (see Figure 24).

6.8.Check Inputs

The user at any time may check whether the numbers entered in the partially solved puzzle are correct. There are two ways to do it:

1) Click the Check button

2) Go to the menu bar and select Action → Check Inputs

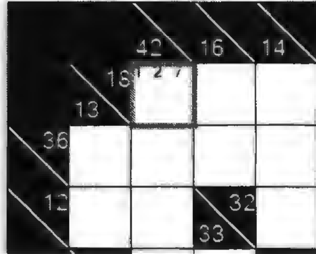


Figure 28. Screenshot: Remove a Choice from the Selected Cell

The application will then solve the puzzle and examine whether the numbers entered by the user are the same as the ones in the answer. If all user-entered numbers match with the ones in the answer, the message shown in Figure 29 is displayed.

Figure 29. Screenshot: Message Displayed when All User Inputs are Correct

When one or more numbers entered are incorrect, the message "One or more inputs are wrong" is displayed, and the input cells that contain the incorrect numbers are highlighted in pink (see Figure 30).

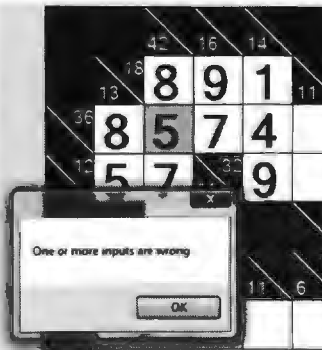


Figure 30. Screenshot: Message Displayed when There are Incorrect Numbers

In case if the application cannot solve the puzzle, the message "The puzzle cannot be solved" will be displayed.

6.9. Clear Inputs

The user can clear all final numbers and the choices that are already entered in the puzzle, by doing one of the following:

1) Click the Clear button
2) Go to the menu bar and select Action → Clear Inputs

The puzzle should then be displayed with all the final numbers cleared.

6.10. Save a Puzzle

At any time when the puzzle is displayed, the user can save the puzzle with final numbers and choices that are currently entered up to this point. The user can also save the empty puzzle or the puzzle that is completely solved. The user can do this in three ways:

1) Click the Save button
2) Go to the menu bar and select Action → Save Puzzle
3) Press Shift + S

The new window (shown in Figure 31) pops up to ask the user to specify the file name and location in which the puzzle should be saved. The user needs to click "OK" after selecting the location and typing in the file name, or click "Cancel" to go back to the main window.

When the user already saves the puzzle in a file and wants to overwrite the file, the user can do so by selecting the file on the location where it is saved. The pop-up message asks whether the user wants to overwrite the existing file, and the user may click "Yes" to overwrite the file.

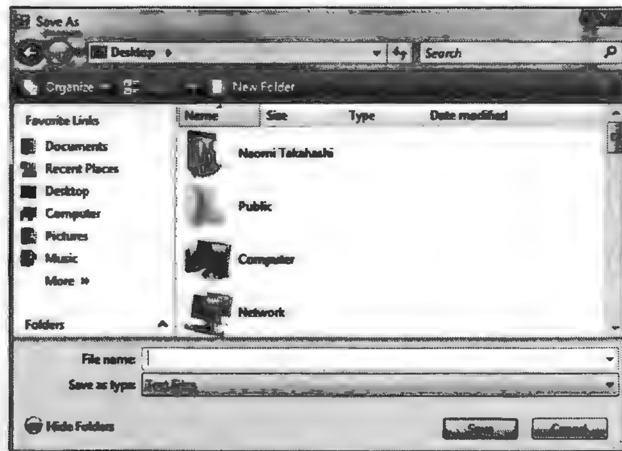


Figure 31. Screenshot: Save a Puzzle in a Text File

6.11. Exit KakuroSolver

The user can exit the application in two ways:

1) Click the Exit button, which is the "X" on the top-right of the main window (see Figure 32)

2) Go to the menu bar and select File → Exit

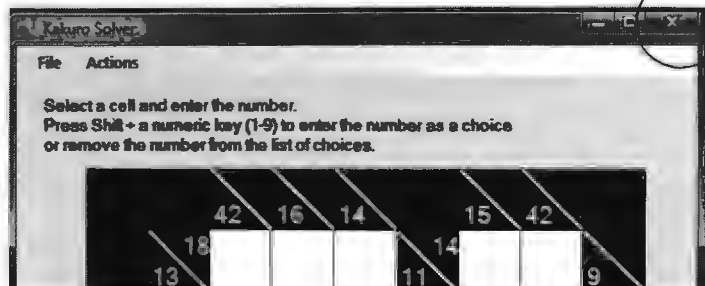


Figure 32. Screenshot: Exit KakuroSolver

BIBLIOGRAPHY

[1] "A Kakuro Solver". http://www.somethinkodd.com/oddthinking/2006/02/12/a-kakuro-solver/. Retrieved on 7/3/09.

[2] Apt, K. R. and Monfroy, E. "Constraint programming viewed as rule-based programming". (2001). Theory and Practice of Logic Programming, 1:6:713-750 Cambridge University Press.

[3] Ben-Haim, Z. "Zvika Ben-Haim – Home Page". http://www.technion.ac.il/~zvikabh/. Retrieved on 7/4/09.

[4] DeVenezia, R. A. et al. "SAS® And Sudoku". In: SAS Global Forum 2007.

[5] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik* 1: 269–271.

[6] Engemann, I. and Wagner, M. "Start of entry: k4kur0/algorithm". http://startofentry.blogdns.org/space/k4kur0/algorithm. Retrieved on 9/12/09.

[7] Han, Y.S., Hartmann, C.R.P., and Chih-Chieh Chen. (September 1993). "Efficient priority-first search maximum-likelihood soft-decision decoding of linear block codes". IEEE Transactions on Information Theory, Volume 39, Issue 5.

[8] Hayes, A. "CodeProject: Set TextBox Height". http://www.codeproject.com/KB/cs/SetTextBoxHeight.aspx. Retrieved on 8/30/09.

[9] "Kakuro.com, the home of Kakuro (cross sums) on the internet". http://kakuro.com/. Retrieved on 7/1/09.

[10] kakuropuzzle.com. "Kakuro Cross Sums". http://www.kakuropuzzle.com/. Retrieved on 10/30/09.

[11] "Kakuro solver". http://kakuro-solver.blogspot.com/. Retrieved on 8/24/09.

[12] Korf, R.E. (1985). Depth-first iterative deepening: an optimal admissible tree search. Artificial Intelligence, 27(1):97-109.

[13] Lerner, A. http://www.cs.tau.ac.il/~alan/Teaching/08b_bioc/exercises/Kakuro_2.pdf. Retrieved on 8/24/09.

[14] Link, J. and Fröhlich, P. (2003). *Unit Testing in Java: How Tests Drive the Code.* Morgan Kaufmann.

[15] Microsoft Corporation. "MSDN Library". http://msdn.microsoft.com/en-us/library. Retrieved on 10/12/09.

[16] Pressman, R. S. (6th ed.) . (2005.). *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill.

[17] Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach.* New Jersey: Prentice Hall.

[18] "SAAM's". http://www.saam007.com/. Retrieved on 7/3/09.

[19] Simonis, H. (October 2005). "Sudoku as a constraint problem". In CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems, p.13-27.

[20] Veeningen, M. "Meilofs website". http://meilof.home.fmf.nl/2008/11/30/a-simple-kakuro-solver/. Retrieved on 7/3/09.

[21] Vion, J. "CSP4J: a black-box CSP solving API for Java". In: Proceedings of the Second International CSP Solver Competition.

[22] Yato, T. and Seta, T. (2003). "Complexity and completeness of finding another solution and its application to puzzles". IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences, E86-A(5):1052–1060.

APPENDIX A. SOURCE CODE - BOARD

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KakuroSolver
{
    //Hold the Kakuro puzzle and find the solution
    public class Board
    {
        Cell[,] cellArray;
        int width, height;

        public Board(int x, int y)
        {
            cellArray = new Cell[x, y];
            width = x;
            height = y;
        }

        //Get the maximum possible number for a run based on the clue
number and the number of cells in the run
        public int getMax(int clue, int numOfCells)
        {
            if (numOfCells == 1)
            {
                return clue;
            }
            else
            {
                int sum = 0;

                for (int i = 1; i < numOfCells; i++)
                {
                    sum += i;
                }
                return (clue - sum);
            }
        }

        //Get the minimum possible number for a run based on the clue
number and the number of cells in the run
        public int getMin(int clue, int numOfCells)
        {
            if (numOfCells == 1)
            {
                return clue;
            }
            else
            {
                int sum = 0;
                int num = 9;
```

```
                for (int i = 1; i < numOfCells; i++)
                {
                    sum += num;
                    num--;
                }
                return (clue - sum);
            }
        }

        //Visit all the clue cells and set the number of input cells
associated with the down clue and/or the across clue of each clue cell
        public void setNumberOfCells()
        {
            for (int i = 0; i < width; i++)
            {
                for (int j = 0; j < height; j++)
                {
                    if (cellArray[i, j].getCellType() == "ClueCell")
                    {
                        ClueCell cell = (ClueCell)cellArray[i, j];
                        if (cell.getAcrossClue() > 0)
                        {
                            int count = 1;
                            for (; i + count < width; count++)
                            {
                                if (cellArray[i + count,
j].getCellType() != "InputCell")
                                {
                                    break;
                                }
                            }
                            cell.setAcrossNumOfCells(count - 1);
                        }
                        if (cell.getDownClue() > 0)
                        {
                            int count = 1;
                            for (; j + count < height; count++)
                            {
                                if (cellArray[i, j +
count].getCellType() != "InputCell")
                                {
                                    break;
                                }
                            }
                            cell.setDownNumOfCells(count - 1);
                        }
                    }
                }
            }
        }

        //Identify impossible numbers from the down clue of a clue cell
and remove the numbers from the possible numbers list of the associated
input cells
        //Precondition: the clue cell has the down clue
        public void setDownPossibleNumbers(ClueCell clue)
```

```
        {
            int x = clue.getX();
            int y = clue.getY();
            InputCell cell;
            List<int> invalidNumbers = new List<int>(9);
            int number;
            int cellsEntered = 0;
            int currentSum = 0;

            //Identify the numbers that are already entered in one of
the associated input cell and add them to the invalid numbers list
            for (int i = 1; i <= clue.getDownNumOfCells(); i++)
            {
                cell = (InputCell)cellArray[x, y + i];
                number = cell.getNumber();
                if (number > 0)
                {
                    invalidNumbers.Add(number);
                    currentSum += number;
                    cellsEntered++;
                }
            }

            int sum = clue.getDownClue() - currentSum;

            //Identify the maximum and minimum possible numbers for the
associated input cells that are still empty
            int max = getMax(sum, clue.getDownNumOfCells() -
cellsEntered);
            int min = getMin(sum, clue.getDownNumOfCells() -
cellsEntered);

            //Add the numbers that are greater than maximum possible
number to the invalidNumbers list
            if (max > 0 && max < 9)
            {
                for (int i = max + 1; i <= 9; i++)
                {
                    if (!invalidNumbers.Contains(i))
                    {
                        invalidNumbers.Add(i);
                    }
                }
            }

            //Add the numbers that are less than minimum possible number
to the invalidNumbers list
            if (min > 1 && min <= 9)
            {
                for (int i = min - 1; i >= 1; i--)
                {
                    if (!invalidNumbers.Contains(i))
                    {
                        invalidNumbers.Add(i);
                    }
                }
```

```
            }

            //Remove the impossible numbers from the possible numbers
list of each empty input cell
            for (int i = 1; i <= clue.getDownNumOfCells(); i++)
            {
                cell = (InputCell)cellArray[x, y + i];
                if (cell.getNumber() == 0)
                {
                    //For each input cell, add 1-9 to the possible
numbers list
                    cell.setPossibleNumbers();
                    List<int> possibleNumbersCell =
cell.getPossibleNumbers();

                    //Remove the impossible numbers from the possible
numbers list of the input cell
                    for (int j = 0; j < invalidNumbers.Count; j++)
                    {
                        possibleNumbersCell.Remove(invalidNumbers[j]);
                    }
                }
            }
        }

        //Identify impossible numbers from the across clue of a clue
cell and remove the numbers from the possible numbers list of the
associated input cells
        //Precondition: the clue cell has the across clue
        public void setAcrossPossibleNumbers(ClueCell clue)
        {
            int x = clue.getX();
            int y = clue.getY();
            InputCell cell;
            List<int> invalidNumbers = new List<int>(9);
            int number;
            int cellsEntered = 0;
            int currentSum = 0;

            //Identify the numbers that are already entered in one of
the associated input cell and add them to the invalid numbers list
            for (int i = 1; i <= clue.getAcrossNumOfCells(); i++)
            {
                cell = (InputCell)cellArray[x + i, y];
                number = cell.getNumber();
                if (number > 0)
                {
                    invalidNumbers.Add(number);
                    currentSum += number;
                    cellsEntered++;
                }
            }

            int sum = clue.getAcrossClue() - currentSum;
```

```
            //Identify the maximum and minimum possible numbers for the
associated input cells that are still empty
            int max = getMax(sum, clue.getAcrossNumOfCells() -
cellsEntered);
            int min = getMin(sum, clue.getAcrossNumOfCells() -
cellsEntered);

            //Add the numbers that are greater than maximum possible
number to the invalidNumbers list
            if (max > 0 && max < 9)
            {
                for (int i = max + 1; i <= 9; i++)
                {
                    if (!invalidNumbers.Contains(i))
                    {
                        invalidNumbers.Add(i);
                    }
                }
            }

            //Add the numbers that are less than minimum possible number
to the invalidNumbers list
            if (min > 1)
            {
                for (int i = min - 1; i >= 1; i--)
                {
                    if (!invalidNumbers.Contains(i))
                    {
                        invalidNumbers.Add(i);
                    }
                }
            }

            //Remove the impossible numbers from the possible numbers
list of each empty input cell
            for (int i = 1; i <= clue.getAcrossNumOfCells(); i++)
            {
                cell = (InputCell)cellArray[x + i, y];
                if (cell.getNumber() == 0)
                {
                    if (cell.getPossibleNumbers() == null)
                    {
                        cell.setPossibleNumbers();
                    }
                    List<int> possibleNumbersCell =
cell.getPossibleNumbers();

                    for (int j = 0; j < invalidNumbers.Count; j++)
                    {
                        //Remove the impossible numbers from the
possible numbers list of the input cell
                        if
(possibleNumbersCell.Contains(invalidNumbers[j]))
                        {

possibleNumbersCell.Remove(invalidNumbers[j]);
```

```
                    }
                }
            }
        }
    }

    //Identify and return the input cell that has least possible
numbers among the empty input cells
    public InputCell getBestCell()
    {
        InputCell bestCell = new InputCell(-1, -1, -1); //Store the
input cell with least possible numbers so far
        int bestNumber = 10;    //Store the number of possible
numbers for the best cell

        for (int j = 0; j < height; j++)
        {
            for (int i = 0; i < width; i++)
            {
                string cellType = cellArray[i, j].getCellType();

                if (cellType == "ClueCell")
                {
                    ClueCell clue = (ClueCell)cellArray[i, j];

                    //Remove the impossible numbers from the
possible numbers list of each associated input cell with the down clue
                    if (clue.getDownClue() > 0)
                    {
                        setDownPossibleNumbers(clue);
                    }

                    //Remove the impossible numbers from the
possible numbers list of each associated input cell with the across clue
                    if (clue.getAcrossClue() > 0)
                    {
                        setAcrossPossibleNumbers(clue);
                    }
                }
                else if (cellType == "InputCell")
                {
                    InputCell input = (InputCell)cellArray[i, j];
                    if (input.getNumber() == 0)
                    {
                        int count =
input.getPossibleNumbers().Count;
                        if (count == 0)
                        {
                            //Puzzle cannot be solved
                            return null;
                        }
                        else if (count == 1)
                        {
                            //Count 1 is always the least; don't
need to visit the rest of the cells because this input cell should be
the best cell
```

```
                                return input;
                            }
                            else if (count < bestNumber)
                            {
                                //If this input cell has less possible
numbers than bestNumber, it must have the least possible numbers up to
this point
                                bestNumber = count;
                                bestCell = input;
                            }
                        }
                    }
                }
            }
            return bestCell;
        }

        //Try to solve the puzzle with the inputs to the point and
return whether the puzzle can be solved
        public bool solve()
        {
            //Identify the input cell with least possible numbers
            InputCell cell = getBestCell();

            if (cell != null)
            {
                if (cell.getNumber() != -1)
                {
                    List<int> possibleNumbers =
cell.getPossibleNumbers();

                    //Try to enter each possible number in the input
cell
                    for (int i = 0; i < possibleNumbers.Count; i++)
                    {
                        cell.setNumber(possibleNumbers[i]);
                        if (solve())
                        {
                            //The rest of the puzzle can be solved
                            return true;
                        }
                    }

                    //All the possible numbers are invalid; the puzzle
cannot be solved, and the wrong number currently entered should be
cleared to go back to one upper level of recursion
                    cell.setNumber(0);
                    return false;
                }
                else
                {
                    //There is no empty input cell; the puzzle has been
solved successfully
                    return true;
                }
            }
```

```
            else
            {
                //There is at least one input cell that has no possible
numbers; the puzzle cannot be solved
                return false;
            }
        }

        //Return the width of the cellArray
        public int getWidth()
        {
            return width;
        }

        //Return the height of the cellArray
        public int getHeight()
        {
            return height;
        }

        //Return the Cell of cellArray from its x and y location
        public Cell getCell(int x, int y)
        {
            if (x >= 0 && x < width && y >= 0 && y < height)
            {
                return cellArray[x, y];
            }
            else
            {
                //Invalid location
                return null;
            }
        }

        //Add cells to cellArray from CreateForm and return whether all
items in cellArray are non-null
        //Precondition: TextBox array must have the same size as
cellArray
        internal bool constructCells(TextBox[,] txt)
        {
            for (int i = 0; i < width; i++)
            {
                for (int j = 0; j < height; j++)
                {
                    string text = txt[i, j].Text.Trim();

                    if (!text.Contains('\\'))    //The cell must be an
input cell
                    {
                        int num;
                        if (text == "")
                        {
                            cellArray[i, j] = new InputCell(i, j);
                        }
                        else if (Int32.TryParse(text, out num))
                        {
```

```
                        cellArray[i, j] = new InputCell(i, j, num);
                    }
                    else
                    {
                        return false;
                    }
                }
                else
                {
                    if (text != "\\")    //The cell must be a clue
cell
                    {
                        string[] clues = text.Split('\\');
                        int across = 0;
                        int down = 0;

                        if (clues[0].Trim() != "" &&
Int32.TryParse(clues[0].Trim(), out down) == false)
                        {
                            return false;
                        }
                        if (clues[1].Trim() != "" &&
Int32.TryParse(clues[1].Trim(), out across) == false)
                        {
                            return false;
                        }
                        cellArray[i, j] = new ClueCell(i, j, across,
down);
                    }
                    else
                    {
                        cellArray[i, j] = new Cell(i, j);
                    }
                }
            }
        }

        //For each input cell, identify the number of associated
input cells for its down clue and/or across clue
        setNumberOfCells();
        return true;
    }

    //Add cells to cellArray from a text file and return whether all
items in cellArray are non-null
    //Precondition: string array must have the same number of items
as the height of cellArray
    internal bool constructCells(string[] lines)
    {
        width = lines[0].Split('|').Length;
        height = lines.Length;
        cellArray = new Cell[width, height];

        string line = "";
        for (int j = 0; j < height; j++)
        {
```

```
                line = lines[j];
                string[] cells = lines[j].Split('|');

                if (cells.Length == width)
                {
                    for (int i = 0; i < width; i++)
                    {
                        int idx;
                        if ((idx = cells[i].IndexOf('\\')) == -1)
//The cell must be an input cell
                        {
                            InputCell input = new InputCell(i, j);
                            if (cells[i].Trim() != "")
                            {
                                int number;

Int32.TryParse(cells[i].Trim().Substring(0, 1), out number);
                                input.setNumber(number);
                                if (cells[i].Contains("("))
                                {
                                    int index = cells[i].IndexOf('(');
                                    string[] choices =
cells[i].Substring(index + 1, cells[i].Length - index - 2).Split(',');
                                    for (int k = 0; k < choices.Length;
k++)
                                    {
                                        int choice;
                                        Int32.TryParse(choices[k], out
choice);
                                        if (choice > 0)
                                        {
                                            input.setChoices(choice);
                                        }
                                    }
                                }
                            }
                            cellArray[i, j] = input;
                        }
                        else
                        {
                            int down, across;
                            Int32.TryParse(cells[i].Substring(0,
idx).Trim(), out down);
                            Int32.TryParse(cells[i].Substring(idx +
1).Trim(), out across);
                            if (down == 0 && across == 0)    //The cell
cannot be a clue cell
                            {
                                cellArray[i, j] = new Cell(i, j);
                            }
                            else
                            {
                                ClueCell clue = new ClueCell(i, j,
across, down);
                                cellArray[i, j] = clue;
                            }
```

```
                        }
                    }
                }
                else
                {
                    return false;
                }
            }

            //For each input cell, identify the number of associated
input cells for its down clue and/or across clue
            setNumberOfCells();
            return true;
        }

        //Clear all the numbers that are entered in the input cells
        internal void clear()
        {
            InputCell input;

            for (int j = 0; j < height; j++)
            {
                for (int i = 0; i < width; i++)
                {
                    if (cellArray[i, j].getCellType() == "InputCell")
                    {
                        input = (InputCell)cellArray[i, j];
                        input.setNumber(0);
                        input.clearChoices();
                        input.setValidity(true);
                    }
                }
            }
        }

        //Return the Board object that is identical as this Board but
all the input cell are empty
        internal Board clone()
        {
            Board clone = new Board(this.width, this.height);
            for (int j = 0; j < this.height; j++)
            {
                for (int i = 0; i < this.width; i++)
                {
                    switch (this.cellArray[i, j].getCellType())
                    {
                        case "ClueCell":
                            ClueCell clue = (ClueCell)this.cellArray[i,
j];
                            clone.cellArray[i, j] = new ClueCell(i, j,
clue.getAcrossClue(), clue.getDownClue());
                            break;
                        case "InputCell":
                            clone.cellArray[i, j] = new InputCell(i, j,
0);
                            break;
```

```
                        case "Cell":
                            clone.cellArray[i, j] = new Cell(i, j);
                            break;
                    }
                }
            }

            //For each input cell of the clone Board, identify the
number of associated input cells for its down clue and/or across clue
            clone.setNumberOfCells();
            return clone;
        }

        //Compare the inputs to this Board with the numbers
        internal bool check(Board answerBoard)
        {
            InputCell input, answer;
            bool isCorrect = true;
            for (int j = 0; j < this.height; j++)
            {
                for (int i = 0; i < this.width; i++)
                {
                    if (this.cellArray[i, j].getCellType() ==
"InputCell")
                    {
                        input = (InputCell)this.cellArray[i, j];
                        answer = (InputCell)answerBoard.cellArray[i, j];
                        int number = input.getNumber();
                        if (number > 0 && number != answer.getNumber())
                        {
                            input.setValidity(false);
                            isCorrect = false;
                        }
                        else
                        {
                            input.setValidity(true);
                        }
                    }
                }
            }
            return isCorrect;
        }
    }
}
```

APPENDIX B. SOURCE CODE - CANVAS

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KakuroSolver
{
    //User control on which the Kakuro puzzle is drawn
    public partial class Canvas : UserControl
    {
        Board board;
        int unitSize = 0;
        protected int selectedX = -1;
        protected int selectedY = -1;
        protected Pen blackPen, whitePen, greenPen;
        protected Font clue, input, choice;
        protected Brush blackBrush, whiteBrush, pinkBrush;

        public Canvas()
        {
            InitializeComponent();
            blackPen = new Pen(Color.Black, 2);
            whitePen = new Pen(Color.White, 2);
            greenPen = new Pen(Brushes.LightGreen, 5);
            blackBrush = Brushes.Black;
            whiteBrush = Brushes.White;
            pinkBrush = Brushes.Pink;
        }

        //Accept the Board and decide the size of cells displayed
        public virtual void addBoard(Board b)
        {
            board = b;
            setUnitSize();
            this.Width = unitSize * board.getWidth();
            this.Height = unitSize * board.getHeight();
            switch (unitSize)
            {
                case 50:
                    clue = new Font(FontFamily.GenericSansSerif, 14);
                    input = new Font(FontFamily.GenericSansSerif, 36,
FontStyle.Bold);
                    break;
                case 35:
                    clue = new Font(FontFamily.GenericSansSerif, 10);
                    input = new Font(FontFamily.GenericSansSerif, 20,
FontStyle.Bold);
                    break;
                case 33:
```

```
                    clue = new Font(FontFamily.GenericSansSerif, 8);
                    input = new Font(FontFamily.GenericSansSerif, 12,
FontStyle.Bold);
                    break;
            }
            choice = new Font(FontFamily.GenericSansSerif, 8);
        }

        //Decide the size of a cell based on the puzzle size
        private void setUnitSize()
        {
            int numOfCells = board.getWidth();
            if (numOfCells < board.getHeight())
            {
                numOfCells = board.getHeight();
            }

            if (numOfCells < 10)
            {
                unitSize = 50;
            }
            else if (numOfCells < 20)
            {
                unitSize = 35;
            }
            else
            {
                unitSize = 33;
            }
        }

        //Return the cell size
        public int getUnitSize()
        {
            return unitSize;
        }

        //Set the location of the cell currently selected by the user
        public void setSelectedLocation(int x, int y)
        {
            selectedX = x;
            selectedY = y;
        }

        //Draw the puzzle
        protected override void OnPaint(PaintEventArgs pe)
        {
            Graphics g = pe.Graphics;
            g.Clear(Color.WhiteSmoke);
            if (board != null)
            {
                if (board.getCell(0, 0) != null)
                {
                    drawCells(g);

                    if (selectedX >= 0 && this.Focused)
```

```
                    {
                        g.DrawRectangle(greenPen, unitSize * selectedX,
unitSize * selectedY, unitSize, unitSize);
                    }
                }
                else
                {
                    drawEmptyCells(g);
                }
            }
            g.DrawRectangle(blackPen, 0, 0, this.Width, this.Height);
        }

        //Draw cells on the puzzle
        protected virtual void drawCells(Graphics g)
        {
            for (int i = 0; i < board.getWidth(); i++)
            {
                for (int j = 0; j < board.getHeight(); j++)
                {
                    Cell c = board.getCell(i, j);
                    float x = i * unitSize;
                    float y = j * unitSize;
                    string type = c.getCellType();
                    if (type == "InputCell")
                    {
                        InputCell cell = (InputCell)c;
                        g.DrawRectangle(blackPen, x, y, unitSize,
unitSize);
                        g.FillRectangle(whiteBrush, x, y, unitSize,
unitSize);
                        if (!cell.getValidity())
                        {
                            g.FillRectangle(pinkBrush, x, y, unitSize,
unitSize);
                        }
                        int num = cell.getNumber();
                        if (num > 0)
                        {
                            g.DrawString(num.ToString(), input,
blackBrush, x, y);
                        }
                        else if (cell.getChoices().Count > 0)
                        {
                            List<int> choices = cell.getChoices();
                            for (int k = 0; k < choices.Count; k++)
                            {
                                int choiceSize = unitSize / 3;
                                if (k < 3)
                                {
                                    g.DrawString(choices[k].ToString(),
choice, blackBrush, x + choiceSize * k, y);
                                }
                                else if (k < 6)
                                {
```

```
                                        g.DrawString(choices[k].ToString(),
choice, blackBrush, x + choiceSize * (k % 3), y + choiceSize);
                                    }
                                    else
                                    {
                                        g.DrawString(choices[k].ToString(),
choice, blackBrush, x + choiceSize * (k % 3), y + choiceSize * 2);
                                    }
                                }
                            }
                        }
                        else
                        {
                            g.DrawRectangle(blackPen, x, y, unitSize,
unitSize);
                            g.FillRectangle(blackBrush, x, y, unitSize,
unitSize);

                            if (type == "ClueCell")
                            {
                                ClueCell cell = (ClueCell)c;
                                g.DrawLine(whitePen, x, y, x + unitSize, y +
unitSize);
                                if (cell.getAcrossClue() > 0)
                                {

g.DrawString(cell.getAcrossClue().ToString(), clue, whiteBrush, x +
unitSize / 2, y);
                                }
                                if (cell.getDownClue() > 0)
                                {

g.DrawString(cell.getDownClue().ToString(), clue, whiteBrush, x, y +
unitSize / 2);
                                }
                            }
                        }
                    }
                }
            }

            //Draw the empty cells on the puzzle
            private void drawEmptyCells(Graphics g)
            {
                for (int i = 0; i < board.getWidth(); i++)
                {
                    for (int j = 0; j < board.getHeight(); j++)
                    {
                        g.DrawRectangle(blackPen, i * unitSize, j * unitSize,
unitSize, unitSize);
                    }
                }
            }
        }
}
```

APPENDIX C. SOURCE CODE - CELL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace KakuroSolver
{
    //Represent a blank cell
    public class Cell
    {
        int x, y;

        public Cell(int xLocation, int yLocation)
        {
            x = xLocation;
            y = yLocation;
        }

        //Return the X location of the cell
        public int getX()
        {
            return x;
        }

        //Return the Y location of the cell
        public int getY()
        {
            return y;
        }

        //Return the type of the cell
        public virtual string getCellType()
        {
            return "Cell";
        }
    }
}
```

APPENDIX D: SOURCE CODE - CELLSIZEFORM

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KakuroSolver
{
    //Form for the user to specify the size of puzzle
    public partial class CellSizeForm : Form
    {
        MainForm mainform;

        public CellSizeForm(MainForm frm)
        {
            InitializeComponent();
            mainform = frm;
        }

        //Accept user inputs (width and height) and pass them to
CreateForm
        private void createButton_Click(object sender, EventArgs e)
        {
            int width, height;
            Int32.TryParse(txtX.Text, out width);
            Int32.TryParse(txtY.Text, out height);

            if (width > 0 && height > 0)
            {
                CreateForm form = new CreateForm(mainform, width,
height);
                form.Show();
                this.Close();
            }
            else
            {
                MessageBox.Show("Enter positive integers for X and Y");
            }
        }

        //Close itself and enable MainForm
        private void cancelButton_Click(object sender, EventArgs e)
        {
            this.Close();
            mainform.Enabled = true;
        }

        //Specify the location of the form so that it appears on the
MainForm
        private void CellSizeForm_Load(object sender, EventArgs e)
```

```
        {
            this.Location = new Point(mainform.Location.X + 50,
mainform.Location.Y + 50);
        }
    }
}
```

APPENDIX E. SOURCE CODE - CLUECELL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace KakuroSolver
{
    //Represent a clue cell
    public class ClueCell : Cell
    {
        int acrossClue, downClue, acrossNumOfCells, downNumOfCells;

        public ClueCell(int xLocation, int yLocation, int across, int
down)
            : base(xLocation, yLocation)
        {
            acrossClue = across;
            downClue = down;
        }

        //Return the across clue of the cell
        public int getAcrossClue()
        {
            return acrossClue;
        }

        //Return the down clue of the cell
        public int getDownClue()
        {
            return downClue;
        }

        //Set how many input cells are in the across run
        public void setAcrossNumOfCells(int num)
        {
            acrossNumOfCells = num;
        }

        //Set how many input cells are in the down run
        public void setDownNumOfCells(int num)
        {
            downNumOfCells = num;
        }

        ///Return how many input cells are in the across run
        public int getAcrossNumOfCells()
        {
            return acrossNumOfCells;
        }

        //Return how many input cells are in the across run
        public int getDownNumOfCells()
        {
```

```
            return downNumOfCells;
        }

        //Return the type of the cell
        public override string getCellType()
        {
            return "ClueCell";
        }

        //Return the cell information as string
        public override string ToString()
        {
            string str = "";
            if (downClue > 0)
            {
                str += downClue.ToString();
            }
            str += "\\";
            if (acrossClue > 0)
            {
                str += acrossClue.ToString();
            }
            return str;
        }
    }
}
```

APPENDIX F. SOURCE CODE - CREATEFORM

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KakuroSolver
{
    //Form for the user to create a puzzle
    public partial class CreateForm : Form
    {
        Board board;
        TextBox[,] textCellArray;
        MainForm mainform;

        public CreateForm(MainForm frm, int x, int y)
        {
            InitializeComponent();
            mainform = frm;
            board = new Board(x, y);
            textCellArray = new TextBox[x, y];
            textCanvas1.addBoard(textCellArray);
            setWindowsSize();
            setButtonsLocation();
            Refresh();
        }

        //Set where the buttons are on the form
        private void setButtonsLocation()
        {
            int x = textCanvas1.Location.X;
            int panelsize = panel1.Width + panel1.Location.X;
            if (textCanvas1.Width < 451)
            {
                x = (this.Width - panelsize - textCanvas1.Width) / 2 +
panelsize;
            }

            textCanvas1.Location = new Point(x, textCanvas1.Location.Y);
            buttonsPanel.Location = new Point((this.Width - panelsize) /
2 - (buttonsPanel.Width / 2) + panelsize, textCanvas1.Location.Y +
textCanvas1.Height + 20);
        }

        //Set the form size based on the puzzle size
        private void setWindowsSize()
        {
            this.WindowState = FormWindowState.Maximized;
            Size maxSize = this.Size;
            this.WindowState = FormWindowState.Normal;
```

```
            if (this.Width < textCanvas1.Location.X + textCanvas1.Width
+ 50)
            {
                this.Width = textCanvas1.Location.X + textCanvas1.Width
+ 50;
            }
            else if (maxSize.Width < textCanvas1.Location.X +
textCanvas1.Width + 50)
            {
                this.Width = maxSize.Width;
            }

            if (this.Height < textCanvas1.Location.Y +
textCanvas1.Height + 50)
            {
                this.Height = textCanvas1.Location.Y +
textCanvas1.Height + 50;
            }
            else if (maxSize.Height < textCanvas1.Location.Y +
textCanvas1.Height + 50)
            {
                this.Height = maxSize.Height;
            }
        }

        //Construct cells as specified by the user
        private void createButton_Click(object sender, EventArgs e)
        {
            if (board.constructCells(textCellArray))
            {
                mainform.addBoard(board);
                mainform.Refresh();
                this.Close();
                mainform.Enabled = true;
            }
            else
            {
                MessageBox.Show("The puzzle contains invalid cell(s).",
"Input error");
            }
        }

        //Close the form and enable the MainForm
        private void cancelButton_Click(object sender, EventArgs e)
        {
            this.Close();
            mainform.Enabled = true;
        }
    }
}
```

APPENDIX G. SOURCE CODE – INPUTCELL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace KakuroSolver
{
    //Represent an input cell
    public class InputCell : Cell
    {
        int number;
        List<int> possibleNumbers;
        List<int> choices;
        bool isCorrect = true;

        public InputCell(int xLocation, int yLocation)
            : base(xLocation, yLocation)
        {
            number = 0;
            setPossibleNumbers();
            choices = new List<int>(9);
        }

        public InputCell(int xLocation, int yLocation, int num)
            : base(xLocation, yLocation)
        {
            number = num;
            setPossibleNumbers();
            choices = new List<int>(9);
        }

        //Set the number in the input cell
        //0 means that there is no number entered
        public void setNumber(int num)
        {
            number = num;
        }

        //Return the number in the input cell
        public int getNumber()
        {
            return number;
        }

        //Add the choice if it is not in the list of choices
        //Remove the choice if it is in the list
        public void setChoices(int choice)
        {
            if (choices.Contains(choice))
            {
                choices.Remove(choice);
            }
            else
```

```
            {
                choices.Add(choice);
            }
        }

        //Return the list of choices
        public List<int> getChoices()
        {
            return choices;
        }

        //Set the initial possible numbers (1-9) for the input cell
        public void setPossibleNumbers()
        {
            possibleNumbers = new List<int>(9);
            possibleNumbers.Add(1);
            possibleNumbers.Add(2);
            possibleNumbers.Add(3);
            possibleNumbers.Add(4);
            possibleNumbers.Add(5);
            possibleNumbers.Add(6);
            possibleNumbers.Add(7);
            possibleNumbers.Add(8);
            possibleNumbers.Add(9);
        }

        //Return the current possible numbers
        public List<int> getPossibleNumbers()
        {
            return possibleNumbers;
        }

        //Return the type of the cell
        public override string getCellType()
        {
            return "InputCell";
        }

        //Return the cell information as string
        public override string ToString()
        {
            string cellString = "";
            if (number > 0)
            {
                cellString += number.ToString();
            }
            if (choices.Count > 0)
            {
                cellString += "(";
                for (int i = 0; i < choices.Count; i++)
                {
                    cellString += choices[i].ToString() + ",";
                }
                cellString = cellString.Substring(0, cellString.Length -
1) + ")";
            }
```

```
            return cellString;
        }

        //Store whether the number currently entered in the cell is
correct
        public void setValidity(bool correct)
        {
            if (correct)
            {
                isCorrect = true;
            }
            else
            {
                isCorrect = false;
            }
        }

        //Return whether the number currently entered in the cell is
correct
        internal bool getValidity()
        {
            return isCorrect;
        }

        //Delete all the choices from the list of choices
        internal void clearChoices()
        {
            choices.Clear();
        }
    }
}
```

APPENDIX H. SOURCE CODE – MAINFORM

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace KakuroSolver
{
    //Form for the user to solve the puzzle
    public partial class MainForm : Form
    {
        Board board, answerBoard;
        int unitSize = 0;
        int selectedX = -1;
        int selectedY = -1;
        //int number = -1;

        public MainForm()
        {
            InitializeComponent();
        }

        //Pass the Board to Canvas
        public void addBoard(Board b)
        {
            board = b;
            canvas1.addBoard(b);
            answerBoard = b.clone();
            unitSize = canvas1.getUnitSize();
            setWindowsSize();
            setObjectsLocation();
            actionsToolStripMenuItem.Enabled = true;
            saveButton.Enabled = true;
            saveToolStripMenuItem.Enabled = true;
            solveButton.Enabled = true;
            checkButton.Enabled = true;
            clearButton.Enabled = true;
            label1.Text = "Select a cell and enter the number.  \nPress
Shift + a numeric key (1-9) to enter the number as a choice \nor remove
the number from the list of choices.";
        }

        //Set the locations of the Canvas and the buttons
        private void setObjectsLocation()
        {
            int margin = this.Width - canvas1.Width;
            canvas1.Location = new Point(margin / 2,
canvas1.Location.Y);
```

```
            buttonsPanel.Location = new Point(this.Width / 2 - 180,
canvas1.Location.Y + canvas1.Height + 20);
        }

        //Set the size of the form
        private void setWindowsSize()
        {
            this.WindowState = FormWindowState.Maximized;
            Size maxSize = this.Size;
            this.WindowState = FormWindowState.Normal;
            int x = this.Location.X;
            int y = this.Location.Y;

            int width = canvas1.Location.X + canvas1.Width + 100;
            if (maxSize.Width < width)
            {
                this.Width = maxSize.Width;
                x = 0;
            }
            else if (this.Width < width)
            {
                this.Width = width;
            }
            else
            {
                //Display the entire description
                this.Width = this.Width + 50;
            }

            int height = canvas1.Location.Y + canvas1.Height + 20 +
buttonsPanel.Height + 70;

            if (maxSize.Height < height + 10)
            {
                this.Height = maxSize.Height - 50;
                y = 0;
            }
            else
            {
                this.Height = height;
                if (maxSize.Height < this.Location.Y + height)
                {
                    y = 0;
                }
            }

            this.Location = new Point(x, y);
            panel1.Size = new Size(this.Width - 20, this.Height - 66);
        }

        //Identify on which cell the user clicks
        private void canvas1_MouseClick(object sender, MouseEventArgs e)
        {
            if (board != null)
            {
                Point point = e.Location;
```

```
                selectedX = point.X / unitSize;
                selectedY = point.Y / unitSize;

                if (board.getCell(selectedX, selectedY).getCellType() ==
"InputCell")
                {
                    canvas1.setSelectedLocation(selectedX, selectedY);
                    canvas1.Refresh();
                }
                else
                {
                    selectedX = -1;
                    selectedY = -1;
                }
            }
        }

        private void createButton_Click(object sender, EventArgs e)
        {
            createPuzzle();
        }

        //Open the CellSizeForm for the user to create a puzzle
        private void createPuzzle()
        {
            if (board != null)
            {
                DialogResult result = MessageBox.Show("Do you want to
save the current puzzle?", "", MessageBoxButtons.YesNoCancel,
MessageBoxIcon.Exclamation, MessageBoxDefaultButton.Button1);
                if (result == DialogResult.Yes)
                {
                    savePuzzle();
                }
                else if (result == DialogResult.Cancel)
                {
                    return;
                }
            }
            CellSizeForm cellsizeform = new CellSizeForm(this);
            cellsizeform.Show();
            this.Enabled = false;
        }

        //Save the current puzzle in the text file
        private void savePuzzle()
        {
            SaveFileDialog saveDialog = new SaveFileDialog();
            saveDialog.DefaultExt = "txt";
            saveDialog.Filter = "Text Files|*.txt";
            saveDialog.AddExtension = true;
            if (saveDialog.ShowDialog() != DialogResult.Cancel)
            {
                string fileName = saveDialog.FileName;
                StreamWriter writer = new StreamWriter(fileName);
```

```
                for (int j = 0; j < board.getHeight(); j++)
                {
                    string line = "";

                    for (int i = 0; i < board.getWidth(); i++)
                    {
                        switch (board.getCell(i, j).getCellType())
                        {
                            case "InputCell":
                                InputCell input =
(InputCell)board.getCell(i, j);
                                line += input.ToString();
                                break;
                            case "ClueCell":
                                ClueCell clue =
(ClueCell)board.getCell(i, j);
                                line += clue.ToString();
                                break;
                            default:
                                line += "\\";
                                break;
                        }
                        line += "|";
                    }

                    line = line.Remove(line.Length - 1);    //remove the
last "|"
                    writer.WriteLine(line);
                }
                writer.Close();
            }
        }

        private void saveButton_Click(object sender, EventArgs e)
        {
            if (board != null)
            {
                savePuzzle();
            }
        }

        private void loadButton_Click(object sender, EventArgs e)
        {
            loadPuzzle();
        }

        //Open the text file that contains the puzzle
        private void loadPuzzle()
        {
            if (board != null)
            {
                DialogResult result = MessageBox.Show("Do you want to
save the current puzzle?", "", MessageBoxButtons.YesNoCancel,
MessageBoxIcon.Exclamation, MessageBoxDefaultButton.Button1);
                if (result == DialogResult.Yes)
                {
```

```
                    savePuzzle();
                }
                else if (result == DialogResult.Cancel)
                {
                    return;
                }
            }
            OpenFileDialog openFileDialog = new OpenFileDialog();
            openFileDialog.Filter = "Text Files|*.txt";
            if (openFileDialog.ShowDialog() != DialogResult.Cancel)
            {
                string fileName = openFileDialog.FileName;
                if (fileName != "" && File.Exists(fileName))
                {
                    string[] lines = File.ReadAllLines(fileName);
                    if (lines.Length > 0)
                    {
                        string[] cells = lines[0].Split('|');
                        int lineCnt = lines.Length;
                        if (lines[lineCnt - 1].Trim() == "")
                        {
                            lineCnt = lineCnt - 1;
                            string[] newLines = new string[lineCnt - 1];
                            lines.CopyTo(newLines, 0);
                            lines = newLines;
                        }
                        board = new Board(cells.Length, lineCnt);

                        if (board.constructCells(lines))
                        {
                            addBoard(board);
                            canvas1.Refresh();
                        }
                        else
                        {
                            MessageBox.Show("The puzzle has incorrect
syntax");
                        }
                    }
                    else
                    {
                        MessageBox.Show("The file is empty");
                    }
                }
                else
                {
                    MessageBox.Show("The file does not exist.");
                }
            }
        }

        //Set the number to the selected input cell
        private void canvas1_KeyPress(object sender, KeyPressEventArgs
e)
        {
            if (selectedX > -1 && selectedY > -1)
```

```
            {
                if (e.KeyChar >= '1' && e.KeyChar <= '9')
                {
                    int number = Int32.Parse(e.KeyChar.ToString());
                    InputCell input = (InputCell)board.getCell(selectedX,
selectedY);
                    input.setNumber(number);
                    canvas1.Refresh();
                }
            }
        }

        //Identify which keys are being pressed and call the
corresponding method
        private void canvas1_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.Control)
            {
                switch (e.KeyCode)
                {
                    case Keys.S:
                        if (board != null)
                        {
                            savePuzzle();
                        }
                        break;
                    case Keys.N:
                        createPuzzle();
                        break;
                    case Keys.O:
                        loadPuzzle();
                        break;
                }
            }
            else if (selectedX > -1 && selectedY > -1)
            {
                if (e.Shift)
                {
                    int choice = 0;
                    switch (e.KeyCode)
                    {
                        case Keys.D1:
                            choice = 1;
                            break;
                        case Keys.D2:
                            choice = 2;
                            break;
                        case Keys.D3:
                            choice = 3;
                            break;
                        case Keys.D4:
                            choice = 4;
                            break;
                        case Keys.D5:
                            choice = 5;
                            break;
```

```
                    case Keys.D6:
                        choice = 6;
                        break;
                    case Keys.D7:
                        choice = 7;
                        break;
                    case Keys.D8:
                        choice = 8;
                        break;
                    case Keys.D9:
                        choice = 9;
                        break;
                }

                if (choice > 0)
                {
                    InputCell input =
(InputCell)board.getCell(selectedX, selectedY);
                    input.setChoices(choice);
                    canvas1.Refresh();
                }
            }
            else
            {
                switch (e.KeyCode)
                {
                    case Keys.Delete:
                        InputCell input =
(InputCell)board.getCell(selectedX, selectedY);
                        input.setNumber(0);
                        input.setValidity(true);
                        canvas1.Refresh();
                        break;
                    case Keys.Right:
                        if (selectedX + 1 < board.getWidth())
                        {
                            selectedX += 1;
                        }
                        canvas1.Refresh();
                        break;
                    case Keys.Left:
                        if (selectedX - 1 >= 0)
                        {
                            selectedX -= 1;
                        }
                        canvas1.Refresh();
                        break;
                    case Keys.Down:
                        if (selectedY + 1 < board.getHeight())
                        {
                            selectedY += 1;
                        }
                        canvas1.Refresh();
                        break;
                    case Keys.Up:
                        if (selectedY - 1 >= 0)
```

```
                    {
                        selectedY -= 1;
                    }
                    canvas1.Refresh();
                    break;
            }
        }
    }
}

private void solveButton_Click(object sender, EventArgs e)
{
    solvePuzzle();
}

//Solve the puzzle and display the solution
private void solvePuzzle()
{
    if (board != null)
    {
        if (board.solve())
        {
            canvas1.Refresh();
        }
        else
        {
            MessageBox.Show("The puzzle cannot be solved.");
        }
    }
    else
    {
        MessageBox.Show("No puzzle is loaded.");
    }
}

private void clearButton_Click(object sender, EventArgs e)
{
    clearPuzzle();
}

//Clear all the user inputs
private void clearPuzzle()
{
    board.clear();
    canvas1.Refresh();
}

private void checkButton_Click(object sender, EventArgs e)
{
    checkPuzzle();
}

//Check whether all the user inputs are correct
private void checkPuzzle()
{
    if (board != null)
```

```
            {
                if (answerBoard.solve())
                {
                    if (board.check(answerBoard))
                    {
                        MessageBox.Show("All inputs are correct.");
                    }
                    else
                    {
                        canvas1.Refresh();
                        MessageBox.Show("One or more inputs are
wrong.");
                    }
                }
                else
                {
                    MessageBox.Show("The puzzle cannot be solved.");
                }
            }
            else
            {
                MessageBox.Show("No puzzle is loaded.");
            }
        }

        private void newToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            createPuzzle();
        }

        private void solveToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            solvePuzzle();
        }

        private void checkToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            checkPuzzle();
        }

        private void clearToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            clearPuzzle();
        }

        private void loadToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            loadPuzzle();
        }
```

```
        private void saveToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            if (board != null)
            {
                savePuzzle();
            }
        }

        //Close the MainForm and exit the application
        private void exitToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            this.Close();
        }
    }
}
```

APPENDIX I. SOURCE CODE - TEXTCANVAS

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KakuroSolver
{
    //User control on which the empty puzzle is drawn and the user
specifies the puzzle
    public partial class TextCanvas : Canvas
    {
        int unitX, unitY, width, height;
        TextBox[,] textCellArray;

        public TextCanvas()
            : base()
        {
            InitializeComponent();
        }

        //Accept the empty puzzle as TextBox array and draw it
        public void addBoard(TextBox[,] textArray)
        {
            textCellArray = textArray;
            width = textCellArray.GetLength(0);
            height = textCellArray.Length / width;  //Length returns the
total number of items in the two-dimensional array
            setUnitSize();
            this.Width = unitX * width;
            this.Height = unitY * height;
        }

        //Draw the empty puzzle
        protected override void OnPaint(PaintEventArgs pe)
        {
            Graphics g = pe.Graphics;
            g.Clear(Color.WhiteSmoke);
            drawCells(g);
            g.DrawRectangle(blackPen, 0, 0, this.Width, this.Height);
        }

        //Decide the size of cells and draw them on the puzzle
        protected override void drawCells(Graphics g)
        {
            Font myFont;
            switch (unitY)
            {
                case 50:
                    myFont = new Font(FontFamily.GenericSansSerif, 14);
```

```
                    break;
                case 35:
                    myFont = new Font(FontFamily.GenericSansSerif, 10);
                    break;
                default:
                    myFont = new Font(FontFamily.GenericSansSerif, 8);
                    break;
            }
            for (int i = 0; i < width; i++)
            {
                for (int j = 0; j < height; j++)
                {
                    float x = i * unitX;
                    float y = j * unitY;
                    g.DrawRectangle(blackPen, x, y, unitX, unitY);
                    g.FillRectangle(whiteBrush, x, y, unitX, unitY);
                    textCellArray[i, j] = new TextBox();
                    textCellArray[i, j].MaxLength = 5;
                    textCellArray[i, j].Font = myFont;
                    textCellArray[i, j].BorderStyle = BorderStyle.None;
                    textCellArray[i, j].Width = (int)(unitX - 14);
                    textCellArray[i, j].Location = new Point((unitX * i
+ 7), ((unitY * (2 * j + 1)) / 2 - 7));
                    this.Controls.Add(textCellArray[i, j]);
                }
            }
        }

        //Decide the cell size
        public void setUnitSize()
        {
            int numOfCells = width;
            if (numOfCells < height)
            {
                numOfCells = height;
            }

            if (numOfCells < 10)
            {
                unitY = 50;
            }
            else if (numOfCells < 20)
            {
                unitY = 35;
            }
            else
            {
                unitY = 33;
            }
            unitX = (int)(unitY * 1.5);
        }

        //Identify on which cell the user clicks and focus the
corresponding TextBox
        private void TextCanvas_MouseClick(object sender, MouseEventArgs
e)
```

```
        {
            Point point = e.Location;
            selectedX = point.X / unitX;
            selectedY = point.Y / unitY;
            textCellArray[selectedX, selectedY].Focus();
        }
    }
}
```