Universiteit Leiden
The Netherlands

Universiteit Leiden
Opleiding Informatica

Solving and Constructing

Kamaji Puzzles

Name: Kelvin Kleijn

Date: 27/08/2018

1st supervisor: dr. Jeanette de Graaf
2nd supervisor: dr. Walter Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Kamaji is a type of puzzle that originated in France. It features a two-dimensional grid filled with numbers that have to be combined in a way such that all the rules of the puzzle are satisfied. First of all, the entries involved in a combination must add up to a given maximum value. Here a combination is a horizontal, vertical or diagonal contiguous series of squares. Exactly one of the entries of the puzzle board contains this maximum value. Secondly, all board entries have to be used in order to solve the puzzle. And finally, all entries that contain the number one can be used any number of times, whereas all other entries can and must be used only once. Like many other puzzles, Kamaji's come in different sizes and have various levels of difficulty. Every Kamaji puzzle is essentially a board of n by n entries where n is an integer, thus all Kamaji boards are square-shaped.

Our aim with this research project is to utilize and examine different strategies to solve Kamaji puzzles with the aid of a computer program and make a qualitative comparison between these strategies. To that end, we ultimately have come up with three different strategies, among others using a SAT-solver.

Contents

1 Introduction

Almost a year has passed since my supervisor, dr. Jeannette de Graaf and I came to discuss potential topics of research for my bachelor thesis. We ultimately stumbled upon a puzzle book that was distributed by *'Denksport'*. Its cover read *Kamaji* which is the name of the puzzle. We scrolled through the book, solved some of the puzzles by hand and became intrigued. We discovered that very little research had been conducted into these puzzles. All we had found was that the puzzle has been used to enhance the problem solving ability of children [Fre] and that it is no longer being distributed. This only strengthened our desire to analyse them further. We noticed that the puzzles were ordered by size and difficulty and wondered whether we could identify factors that underpin the difficulty of a given puzzle. How can we effectively solve Kamaji puzzles? Is it possible to construct puzzles of a given level of difficulty? We set out to develop different strategies to search for solutions to Kamaji puzzles. This thesis is the result of our research.

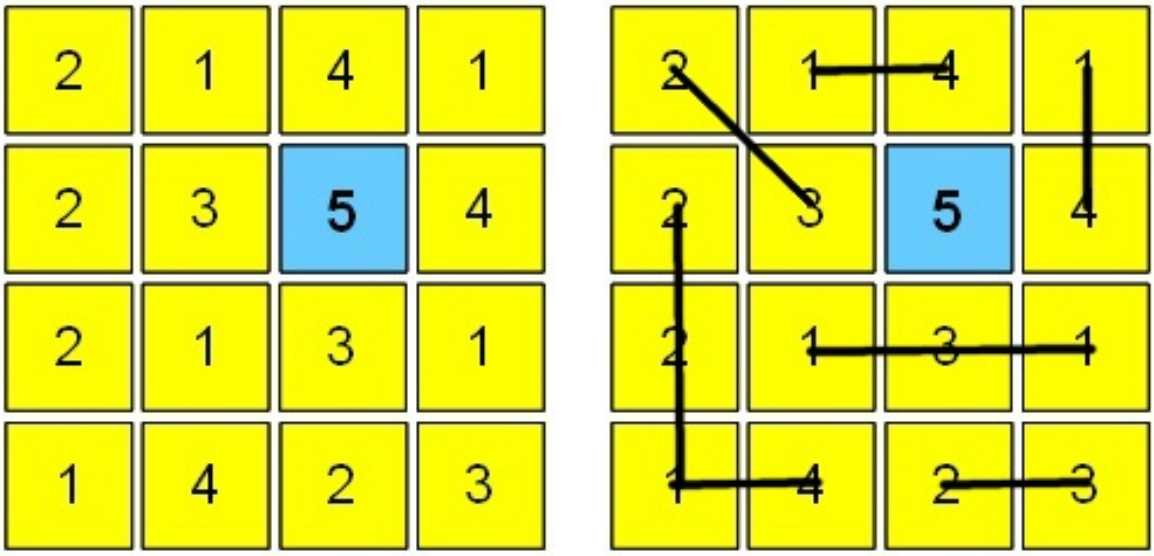
Sadly, very little research has been conducted into this puzzle. There is, however, a substantial amount of research into some puzzles that are similar to the Kamaji puzzle. Two examples are the Japanese puzzles Sudoku and Kakuro. In [OL06] a method to reduce Sudoku puzzles to an instance of SAT is described extensively. Another example of a puzzle that can be reduced to SAT is the binary puzzle [Bia12]. This has inspired us to construct an algorithm that reduces a given Kamaji puzzle to an instance of SAT, and we have succeeded.

In this thesis we will first provide a detailed description of the game and its rules and we will present concrete examples. After that comes a brief section dedicated to explaining the framework and the implementation that we have used. Next, in Section 3 we will give a thorough description of each of the three solution search strategies in separate subsections. The solution search strategies are all implemented in a computer program that we wrote in C++. We have also added a short note on SAT, the Boolean satisfiablity problem and an introduction to MiniSAT, the SAT-solver that we have used to implement the third solution search strategy. We will make a comparison between Kamaji puzzles of various levels of difficulty. In Section 6 we discuss the experiments that we have run, comparing the performance of the solution search strategies among others. Section 7 explains the framework. We conclude in Section 8, also mentioning future work.

This thesis is the result of a bachelor project at Leiden Institute of Advanced Computer Science (LIACS), Leiden University, supervised by dr. J. de Graaf and dr. W. Kosters.

2 Introduction to Kamaji

Kamaji puzzle boards are n by n in size. Figure 1(a) shows a sample puzzle board, where $n = 4$. The board contains several yellow-coloured squares that contain integers. All the numbers in the grid must be at least 1. Thus, all entries must be non-negative and zero is not a valid entry. The puzzle board features a single square that stands out as it is coloured blue instead of yellow. The blue-coloured square represents a special value that is larger than all the other numbers on the board. From here on we will refer to this special value as the *"Maximum Value"*. On most puzzle boards, the number of entries along the two dimensions (the value of n in "n by n board") is equal to the Maximum Value, although this is not one of the requirements that a puzzle board must satisfy in order to qualify as a Kamaji. The Kamaji puzzle that is featured in Figure 1 does not have this property. In this Kamaji, the Maximum Value is 5 while the size of the board's dimensions is equal to 4. There are no restrictions on the position of the Maximum Value within the board. It can be in one of the board's outer rows or columns as well as somewhere in the middle of the board. Now that we have discussed all of the conditions regarding the board's dimensions and the values that the board's entries contain let us discuss the rules and how one can solve a Kamaji puzzle.



(a) Puzzle Board (b) Solution

Figure 1: A 4×4 Kamaji Puzzle And Its Solution

In order to solve a Kamaji, one must combine adjacent entries of the puzzle such that for every combination, the numbers that are covered by this combination add up to the Maximum Value. Combinations can only be made along straight lines, horizontally, vertically or diagonally. Every entry must be used at least once and all entries that contain numbers that are greater than 1 must be used exactly once. When one solves a Kamaji by hand, one can cover sets of entries that add up to the Maximum Value. This is exemplified in Figure 1(b), which shows a solution to the puzzle in Figure 1(a). Notice that this solution is not unique as the entry in the bottom row that contains the number 4 can also be combined with the entry in the row above it, while all other combinations remain the same. All numbers are covered by precisely one combination except the one in the lower-left entry of the grid. That number is covered by two combinations. This is legal, because the lower-left entry contains the number 1.

There are some simple, yet effective strategies that one can use to solve a given Kamaji puzzle. They are the most obvious methods to use when solving a Kamaji puzzle by hand:

1. Seeking a solution by first considering the numbers that are in the corners of the Kamaji, that is the upper-left, upper-right, lower-left and lower-right entries of the puzzle. If for any of these entries, there is only one possible way to combine the number with its neighbours in a specific direction, one can draw a line covering the entries that are included in this combination. If the afore-mentionted scenario emerges, it is certain that the combination that was found, must be a part of every solution to the puzzle, if any such solution exists at all. The entries in the corners of the puzzle have fewer neighbouring entries. Thus, there are less possibilities and one is more likely to find an entry for which only one valid combination exists in the corners of the puzzle.

2. Seeking a solution by trying to cover the entries that contain the number that is one less than the Maximum Value first. So, if m denotes the Maximum Value of a given puzzle, then we first try to make combinations starting with the entries that are equal to $m - 1$. The only way to form a combination with these entries is by combining them with an adjacent entry that contains the number one. Once we have tried to cover all these entries, it may occur that not all of them can be combined in only one way, because some of these entries are surrounded by multiple ones. In that case, we can proceed by trying to cover all other entries that contain $m - 1$. Then we move on to the entries that contain $m - 2$, then to entries that contain $m - 3$, and so forth, until we find a solution or discover that none exists. This approach is the foundation for the second solution search strategy *Biggerfirst*.

The two strategies that we have described so far are both based on straightforward observations and can be used repeatedly. The first strategy makes sense for two reasons. First of all, the lower the number of neighbouring entries an entry can potentially be combined with, the greater the probability that there is only one possible way to combine this entry. The most extreme case of this principle occurs when for some entry there is only one neighbouring entry that it can possibly be combined with, in which case it is certain that the entry must be combined with that particular neighbour to obtain a solution to the puzzle, if any solution exists. Secondly, the lower the number of neighbouring entries for a given entry, the less time it generally takes to verify whether or not there is only one possible combination that can cover this entry. The second strategy accounts for the bigger numbers in the grid first. Bigger numbers generally make for shorter combinations, that is, combinations involving fewer entries. The bigger a number is, the more likely it is that the sum of the entry and its neighbour will exceed the Maximum Value. This in turn leads to a higher probability that there is only one way to combine this entry. Thus, even though one may still have to try to form combinations with several neighbouring entries, this approach too has its advantages.

We now provide two definitions that we will use throughout the text:

Definition 1: *In the context of a given Kamaji puzzle, a* ***piece*** *is a combination of entries of the puzzle such that there exists an ordering of the entries such that each subsequent entry in the ordering is adjacent to its predecessor and the values contained by the entries that the piece covers, add up to the given Maximum Value.*

Definition 2: *In the context of a given Kamaji puzzle, a* ***solution*** *is a set of pieces such that for each piece that is in the set, the values that are contained by the entries that the piece covers, add up to the given Maximum Value, each of the puzzle's entries that contains a number greater than one is covered by exactly one of the pieces from the set of all pieces and each of the puzzle's entries that contains a one is covered by one or more pieces from the set of all pieces.*

3 Solution Search Strategies

In this section we will describe the three solution search strategies that we have implemented and used to find solutions to given Kamaji puzzles. First we will describe how straightforward Brute Force Search can be applied to solve puzzles. Then, we will describe a strategy called *Biggerfirst*. This strategy seeks a solution by accounting for the largest values on the board repeatedly. This is the second strategy that was discussed in the previous chapter and in some cases its application has to be followed by application of the Brute Force Search approach in order to yield a solution. Lastly, we will describe how one can solve a given Kamaji puzzle by reducing the problem of finding its solution to solving an instance of the Boolean satisfiability problem, also known as SAT.

3.1 Brute Force Search Approach

"Brute Force Search" is a problem-solving technique that is commonly used to search for solution candidates in combinatorial problems [Ber81]. We apply Brute Force Search starting with the entry in the upper-left corner of the grid. The operation of Brute Force Search can be roughly described as follows:

Starting from the first unused entry of the grid in order from left to right and from top to bottom, we try to form combinations of entries that have values that add up to the Maximum Value in four different directions. The four directions we expand combinations in are: 1) right, 2) down-right, 3) down and 4) down-left. We try these directions in the order we have stated them here. That is, if we can not find a valid combination by expanding in a certain direction, we try to find one by expanding in the next direction in the given order. To keep track of the partial solution that has so far been constructed, we use an extra two-dimensional grid. If we find a combination, we proceed recursively to the next available entry on the grid. In this context, available means that the entry either is not included in any combination so far or that it contains the number one. If there is no such entry, that means we have found a solution and we save this solution. If there is an available entry, we repeat the process all over starting from this new entry.

Note that Brute Force Search does not need to abort immediately after it has found a solution. As will be stated also in Section 7 , the framework provides two variants of the Brute Force Search approach. One of them applies Brute Force Search to seek a solution and will continue until it has found all of the existing solutions, though it will first store the solution it finds first as the original solution. Each time it finds a solution, it will print a two-dimensional grid to represent it. After it has finished its operation, it will show the original solution to the user again. The other variant follows the same approach, but it will abort after it has found and stored the first found solution, if any exists. Thus, if no solution exists to a given puzzle, both variants of the Brute Force Search approach will carry out the same computations and will do so in the exact same order.

It is very important to note that the solution representations that Brute Force Search yields for a given puzzle are not neccesarily distinct. Solution representations can be identical and still represent different solutions. Of course, this observation is only relevant for the variant of Brute Force Search that continues to seek for solutions after it has found the first one. For instance while a puzzle may have two solutions, Brute Force Search might generate the same solution representation twice, where one of the representations represents one of the two solutions of the puzzle and the other representation represents the other one. It must therefore be pointed out that, from a mathematical perspective, there exists a one-to-many relation between the set of solution representations that are produced by the Brute Force Search Approach and the complete set of solutions to a puzzle as represented by sets of pieces. Since the second solution search strategy, *Biggerfirst*, sometimes invokes Brute Force Search as a part of its operation, this strategy too may produce solution representations that are identical and still represent different solutions. *Biggerfirst* will be discussed in the next subsection, but first we will show an example of how Brute Force Search can yield the same solution representation multiple times and how this representation can map to distinct solutions of the puzzle.

Consider once more the puzzle displayed in Figure 1. As stated before, the solution posed in Figure 1(b) is not the only solution. The other solution can be obtained by combining the number 4 in entry $(3, 2)$ (the right neighbour of the lower-left entry) with its upper neighbour, entry $(2, 1)$ instead of the one to its left. Brute Force Search will find both solutions for this puzzle and it will represent both solutions by the following two-dimensional array:

01 -2 02 -3

04 01 08 03

04 -5 05 -5

-4 06 07 07

In this representation, entries that contain the same absolute value are covered by the same piece. When a new piece is put on the board, the entries that are covered by the piece and contain a one in the puzzle will be assigned the negation of the number of that piece and the entries that contain a number > 1 in the puzzle will be assigned the number itself. For instance, the third piece that was put, covers entries (0,3) and (1,3); (0,3) contains one and therefore gets -3, and (1,3) contains 4 and gets 3. The only exception to this rule occurs if an entry that contains the number one has already been used and is used again to add a new piece. If a piece denoted by the number p is put on the board, only the yet unused entries covered by this piece will be assigned $-p$ or p depending on whether the corresponding puzzle board entry contains a one or a number greater than one. The only difference between the two solutions of the puzzle concerns the placement of the sixth piece. Entry (3,1) can form a piece with either the entry above it or the one to its left. Because both these entries have been used already, they will not be assigned -6 upon placement of the sixth piece in the solution representation board. Hence, there is no way to tell whether the given solution representation represents one solution or the other. That being said, Brute Force yields exactly one solution representation for each distinct solution.

3.2 Biggerfirst Search

The second solution search strategy that we have implemented is *Biggerfirst*. As suggested by this strategy's name, this algorithm starts by taking the entries that contain large values into consideration first. As mentioned before, the entry that contains the Maximum Value forms a combination by itself. Once the entry containing the Maximum Value has been accounted for, the entries containing the second biggest number are those that contain the Maximum Value minus one. *Biggerfirst* first considers all entries that contain the Maximum Value minus one. If only one potential combinaton exists for any such entry, then if any solution to the puzzle exists, it must contain this combination. Therefore, *Biggerfirst* will structure an initial candidate solution that includes all these combinations. Next, the algorithm considers all entries that contain the value that is one less and, again, for each unused entry will form a combination if only one such combination exists. Then it repeats this for the entries containing the next biggest value in the grid and so forth, until all entries including those that contain 1 have been considered. This is the main process of operation of *Biggerfirst* and is referred to as a "run". *Biggerfirst* will repeat this process until the end of the first run that yields no new combinations. Then the remaining unused entries are accounted for by application of the Brute Fore Search Strategy described in the previous section. In general, puzzles that have a unique solution can mostly be solved entirely by application of the *Biggerfirst* solution search strategy, and we will prove that if *Biggerfirst* can solve a given puzzle without the application of the Brute Force Search strategy, the puzzle must have a unique minimal solution.

Note: *By a "minimal" solution to a Kamaji puzzle, we mean a solution such that no strict subset of the set of pieces that represents it also represents a solution.*

Claim: *If Biggerfirst can solve a given puzzle without the additional application of the Brute Force strategy, the puzzle has a unique minimal solution: the solution found by Biggerfirst.*

Suppose we apply the Solution Search strategy *Biggerfirst*, as described above, to solve a given Kamaji puzzle and suppose that by the mere application of *Biggerfirst* we find a solution to the puzzle. The claim essentially states that these two suppositions imply that the given puzzle has a unique solution. As previously stated, any solution to a given puzzle can be represented in abstract form as a set of pieces. First of all, the solution that *Biggerfirst* yields can be visualised as a set of pieces that were added to the set one at a time, because *Biggerfirst* considers the entries that have not yet been covered by any piece, one at a time. Therefore, the pieces belonging to the solution that *Biggerfirst* finds, were added in a specific chronological order. Thus, one of the resulting pieces was added first to the final set of pieces in the process of searching a solution. When the operation of *Biggerfirst* begins, the set of pieces representing the partial solution is the empty set, since no pieces have yet been laid. That means that the set containing zero combinations (e.g., the empty set) is a subset of every solution set.

Another relevant observation is that in the process of searching for a solution for a given puzzle using *Biggerfirst*, adding a piece can only lead to a reduction of the number

of potential pieces that cover any of the entries that remain unused after adding the piece. This implies that if only one piece exists that can cover a given entry when the candidate solution is still empty, there can never exist a different piece that covers this entry. Therefore, any set of pieces that represents a solution must contain it. With these observations in place, we can now apply the principle of structural induction to prove the claim that was stated before.

Proof

Step1: *Suppose Biggerfirst is applied to a given puzzle and solves it entirely. This implies that a non-empty set of sets of pieces that represent a solution to the puzzle exists. This can be referred to as* ***The solution set****.*

Step2.1: *The empty set is a subset of every set and thus, a subset of all sets of pieces that represent a solution to the given puzzle.*

Step2.2: *In chronological order, the first piece that was added by Biggerfirst was added to cover an entry that otherwise could not be covered by any piece, given the empty set of pieces as the current partial solution. This implies that in any case, this is the only piece that can cover this specific entry. Therefore, every set of pieces that represents a solution to the puzzle must include the piece that was addeds first by Biggerfirst*

Step3: *Suppose that at some point in time during the operation of Biggerfirst, a partial solution has been constructed. Suppose furthermore that the set of pieces that represents this partial solution is a subset of every set of pieces that represents a complete solution to the given puzzle. Because, every solution contains this partial solution set and adding pieces to this set can only lead to a reduction of the number of pieces that any yet unused entry can be covered by, if Biggerfirst lays a successive piece, this piece contains some entry that can only be covered by this piece. Therefore, if every solution contains the partial solution as a subset, then the union of this subset and the piece that will be laid next by Biggerfirst is also a subset of every solution to the puzzle.*

Step4: *Thus, for any given Kamaji puzzle, a set of pieces that represents a solution to this puzzle and was found by Biggerfirst must be a subset of every set of pieces that represents a solution to the puzzle. Adding additional valid pieces, if any such exist, will yield a solution that is non-minimal as these combinations can obviously be left out. This completes the proof.*

□

It is important to note that a solution that is found by *Biggerfirst* is said to be ***minimal***. Let us clarify the difference between a ***unique solution*** and a ***unique minimal solution*** by means of an example. Suppose that we execute *Biggerfirst* with the following puzzle as input:

4	2	1	1
2	1	1	2
1	1	2	3
1	2	3	2

Figure 2: 4×4 puzzle

Biggerfirst will solve this puzzle in a single run. The combinations that *Biggerfirst* finds during its first run are illustrated below, where the leftmost image shows the partial solution after the number 3 has been covered, the middle image shows the partial solution after the number 2 has been covered and the rightmost image shows the solution that is found after all 1's have been covered. We ignore the upper-left entry, which contains the Maximum Value, because it is not relevant for the intended purpose of this demonstration.

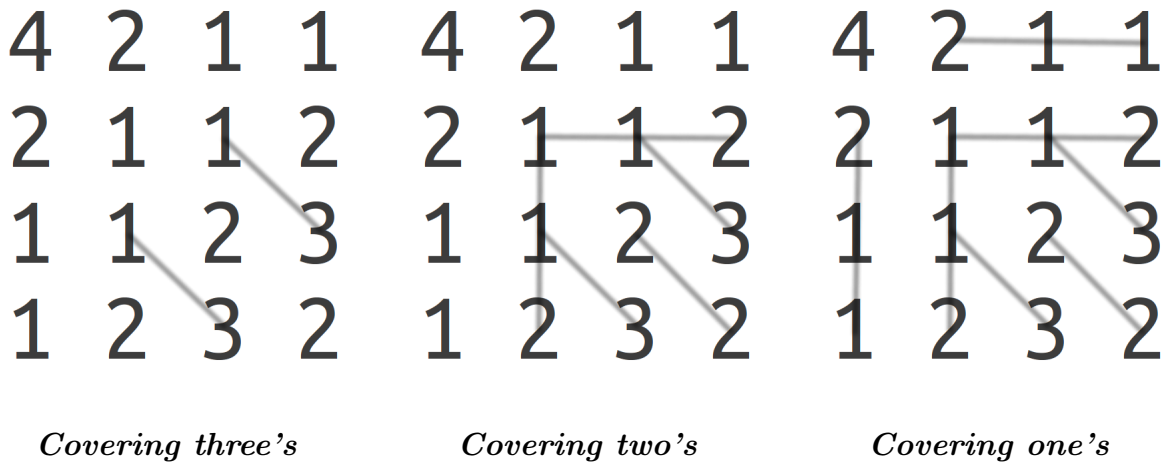


Figure 3: *Biggerfirst* in operation (from left to right)

Even though the rightmost image represents a solution to the puzzle, we can still add a piece that covers all the one's along the diagonal. Note that while this would contribute nothing to the solution, since we already have one, it is not illegal to do so. Nonetheless, it would yield a solution that is distinct from the one that was found by *Biggerfirst*. We therefore say that *Biggerfirst* has found a ***unique minimal solution.***

3.3 Reduction to SAT

The third strategy encompasses both the application of the SAT-solver MiniSAT and the reduction of the Kamaji puzzle to an instance of SAT in DIMACS/CNF format to enable the use of MiniSAT. In this section we will provide a detailed account of the reduction of a Kamaji puzzle to an instance of SAT. First of all, we briefly describe the minimal set of requirements that the result of a reduction must satisfy and how this is reflected in the reduction procedure to assure that a solution to the generated instance of SAT can be mapped back to a unique solution of the puzzle. We have implemented a function that produces an instance of SAT specific to a given Kamaji. In addition, we have implemented a function that takes a solution produced by MiniSAT as input along with other input files and translates this to a human-readable solution of the Kamaji puzzle. In the first subsection, we briefly introduce SAT, the Boolean satisfiability problem and discuss the term CNF. In the second subsection we discuss the DIMACS/CNF format and provide a simple example of an instance of SAT in DIMACS/CNF format. Then, we will provide a detailed description of the process that we use to reduce a two-dimensional grid representation of any Kamaji puzzle to a corresponding SAT instance in DIMACS/CNF format.

3.3.1 Introduction to SAT

The Boolean satisfiability problem, also known as SAT, is the problem of finding a satisfying assignment of the variables in a logic formula that is structured in conjunctive normal form (CNF in short). The problem is well-known among computer scientists and it is the first problem that has ever been proven to be NP complete [Coo71]. This means that all computational problems that are in the nondeterministic polynomial time complexity class NP can be reduced to SAT. An instance of SAT is simply a logic formula in conjunctive normal form.

Conjunctive normal form refers to a certain way in which logic formulas can be structured. A logic formula is in conjunctive normal form if it is a Boolean expression and is structured as a conjunction of clauses where each clause is a disjunction of literals. An example is the formula F shown below:

$$F \equiv (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4 \vee x_2)$$

The formula displayed above will evaluate to true if and only if both clauses evaluate to true, which is the case if each clause contains at least one literal (a variable or its negation) that holds true. For the simple two-claused CNF formula above, there are four variables yielding 2 to the power of 4 different assignments of which only some evaluate formula F to True. Two such assignments are

1. $x_1 = \texttt{true}$, $x_2 = \texttt{true}$, $x_3 = \texttt{true}$, $x_4 = \texttt{false}$.
2. $x_1 = \texttt{true}$, $x_2 = \texttt{true}$, $x_3 = \texttt{false}$, $x_4 = \texttt{true}$.

In the third subsection, we will explain how to convert a Kamaji puzzle to an appropriate CNF formula, but first we will shed some light on the structure and syntax rules of the DIMACS/CNF format.

3.3.2 The DIMACS/CNF Format

The DIMACS/CNF format allows for the specification of any instance of SAT and provides an intuitive syntax for describing a logic formula in conjunctive normal form. It does not come with a unique file extension and we have simply saved all our DIMACS/CNF translations in plain text files (.txt file extension). Below this text, one can find the contents of a file containing a simple CNF formula in DIMACS/CNF format, see Figure 4.

The first two lines start with the symbol 'c'. In the DIMACS/CNF format, the symbol 'c' indicates the start of a comment. So, the first two lines of the file are comments. As a convention, the first line that is not a comment starts with the symbol 'p' and is followed with the phrase 'cnf'; here, this is followed by the numbers 3 and 2, respectively. The term 'cnf' serves as a hint to the SAT-solver that will read this file that what follows should be interpreted as a formula in conjunctive normal form. The first number (in this case 3) denotes the number of variables the formula contains and the second number (in this case 2) denotes the number of clauses. Although it is good style to document the correct number of variables and clauses, this is not neccessary for the correct operation of MiniSAT. All the following lines, in this case line 4 and line 5, provide the structure of the formula in conjunctive normal form. Each line represents a separate clause. As a rule of syntax, each line is ended by the number zero and all the numbers on the same line that precede it represent literals of the clause. The values 1 and -1 represent variable x_1, 2 and -2 represent x_2 and 3 and -3 both represent x_3, where negative numbers denote negation. Hence, the formula represented in Figure 4 is given by: $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$

```
c simple_v3_c2.cnf
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

Figure 4: A simple formula in DIMACS/CNF format

3.3.3 Translation Procedure

In the previous subsection we have briefly introduced the DIMACS/CNF format and we have seen how its syntax is quite intuitive for the purpose of describing logic formulas in conjunctive normal form. In order to translate a given Kamaji puzzle to a file similar to the one described in the previous subsection, we need a method to convert a puzzle represented by a two-dimensional grid into a corresponding CNF formula. Furthermore, we need a method to translate the satisfying assignment found by the SAT-solver back to a set of pieces such that this set represents a solution to the puzzle.

We observe that one can solve a puzzle by virtually laying pieces onto the puzzle board until the set of all pieces that were laid represents a solution. Also observe that for each piece in a solution to a puzzle, each entry of the puzzle board is either covered by it or it is not. These observations allow us to determine what the variables in our to be constructed CNF formula will represent. First, we generate all the existing pieces for the given puzzle board. From here on, we will use p to denote the number of pieces that were found. A variable x_{ijk} in our translation will correspond to a specific entry and a specific piece, where i and j specify the entry and k specifies the piece, with $1 \leq k \leq p$ (p denotes the total number of pieces). If x_{ijk} is true for some i, j and k in a solution produced by MiniSAT, this means that the solution contains the piece that is represented by k and that entry (i, j) is covered by it. If x_{ijk} is false for some i, j and k that means that entry (i, j) is not covered by piece k and piece k is not part of the solution.

Note that $n \cdot n \cdot p$, with the puzzle size being $n \times n$, constitutes an upper bound for the number of variables that the CNF formula will contain. A piece, however, can only cover a strict subset of all puzzle entries. Therefore, there exist variables x_{ijk} such that piece k cannot cover entry (i, j) and are thus false. Because those variables will always evaluate to false, they are not relevant for the reduction. For that reason, for any entry (i, j), we will only include variables x_{ijk} in our formula such that piece k actually covers entry (i, j).

The translation from puzzle to formula consists of three components. We will now describe each of them in detail by using Figure 1 as example.

First Component. In order to obtain a solution to the puzzle, every entry must be covered by at least one piece. More specifically, all entries that contain a value greater than one must be covered by exactly one piece and all entries that contain the number one must be covered by at least one piece. Since every variable represents a pair of entry and a specific piece that covers this entry, we can enumerate all variables corresponding to an entry for every entry. It is possible to determine for each entry of the puzzle board which pieces can cover that entry. Let us take entry $(1, 1)$ of Figure 1 as an example. There are four distinct pieces that could cover this entry, these are pieces 1, 3, 6 and 8. We then have variables x_{111}, x_{113}, x_{116} and x_{118}, one for each piece that potentially covers entry $(1, 1)$. To enforce this logic, we will need to add the following clause:

$$(x_{111} \vee x_{113} \vee x_{116} \vee x_{118})$$

Note that at least one of the variables has to evaluate to true in order to satisfy the clause. This translates to: at least one piece has to cover entry $(1, 1)$. For each entry in the puzzle, we will include a clause that is a disjunction of all the variables that are associated with that entry. This process constitutes the first component of the translation procedure.

Second Component. The entries that contain a number that is greater than one must be covered by exactly one piece. This requirement must somehow be reflected in the CNF formula and its solution. The second component of the translation methodology is designed to address exactly this. Consider once more entry $(1, 1)$ of Figure 1. This entry contains the number three. Thus, it can not be covered by all of the four pieces that could potentially be put there. In fact, if the rules of the Kamaji puzzle are to be respected, it can and must only be covered by exactly one of the four pieces. That means, precisely one of the four variables that make up the clause shown in the previous paragraph must be true and all others must be false. We can make sure of this by adding one clause for each distinct pair of variables corresponding to the given entry $(1, 1)$ and negate the variables. We then get the following additional clauses:

1. $(\neg x_{111} \vee \neg x_{113})$
2. $(\neg x_{111} \vee \neg x_{116})$
3. $(\neg x_{111} \vee \neg x_{116})$
4. $(\neg x_{113} \vee \neg x_{116})$
5. $(\neg x_{113} \vee \neg x_{118})$
6. $(\neg x_{116} \vee \neg x_{118})$

If any two or more of the variables corresponding to entry $(1, 1)$ are true, one of the clauses above will evaluate to false as both of its corresponding negative literals will be false. Therefore, the entire CNF formula will evaluate to false, because the formula will be true if and only if all of its clauses hold true. In other words, all of the six clauses displayed above will only be true if one of the variables is true or none of them are. This is equivalent to the statement that at most one of the variables is true. And since the clause $(x_{111} \vee x_{113} \vee x_{116} \vee x_{118})$ is true only if at least one of the variables is true, we can be sure that precisely one will be true if all of the clauses are to evaluate to true. The number of clauses, all of which have a length of two, that this component will add for each entry that contains a value greater than one is given by:

$$(\ell - 1) + (\ell - 2) + (\ell - 3) + \ldots + 1 = \sum_{i=1}^{\ell-1} i = \frac{\ell \cdot (\ell - 1)}{2}$$

where ℓ denotes the number of pieces that are associated with the given entry. With all that being said, it must be kept in mind that such a set of additional clauses must be added to the formula only for those entries of the puzzle that contain a number that is greater than one.

Third Component. The afore-described first and second component of the reduction procedure do not suffice for the purpose of translating a Kamaji puzzle into a logic formula in conjunctive normal form. Consider once more entry $(1, 1)$ of Figure 1. As stated before, this entry can be covered by one of four pieces: 1, 3, 6 or 8. The third component is based on the observation that a set of variables that corresponds to the same piece must either all be true or all are false. In our example piece 3 happens to cover the entries $(0, 1)$, $(1, 1)$ and $(2, 1)$. Thus, the variables that are associated with piece 3 are x_{013}, x_{113} and x_{213}. We then get the following logic proposition:

$$x_{013} \iff x_{113} \iff x_{213}$$

In order to enforce this logic, we must ascertain that for each piece, a set of clauses is added to the resulting CNF formula such that either all variables corresponding to the piece are true or they are all false. Thus, the following must hold:

$$(x_{013} \wedge x_{113} \wedge x_{213}) \vee (\neg x_{013} \wedge \neg x_{113} \wedge \neg x_{213})$$

Let us start with the most simple case. How can we make sure that two variables are both true or both false? Let us use the variables x_{111} and x_{113} to exemplify the method. For these two variables we can add the following two clauses:

1. $x_{111} \vee \neg x_{113}$
2. $x_{113} \vee \neg x_{111}$

Then if x_{111} is true, x_{113} must be true from clause 2, and if x_{113} is true, x_{111} must be true as well from clause 1 in order to satisfy both clauses. So x_{111} and x_{113} are either both true or both are false. We can extend this idea to any number of variables by adding two clauses for each distinct pair of variables that are associated with a given piece. For this example, we therefore add all of the following clauses for piece 3:

1. $x_{013} \vee \neg x_{113}$
2. $x_{113} \vee \neg x_{013}$
3. $x_{013} \vee \neg x_{213}$
4. $x_{213} \vee \neg x_{013}$
5. $x_{113} \vee \neg x_{213}$
6. $x_{213} \vee \neg x_{113}$

The number of clauses that the third component will add for a given piece depends on the number of entries that the piece covers. If l denotes the number of entries that a piece k covers, then the number of clauses added for piece k is given by:

$$2 \cdot ((\ell - 1) + (\ell - 2) + \ldots + 1) = 2 \cdot \sum_{i=1}^{\ell-1} i = \ell \cdot (\ell - 1)$$

This completes the translation procedure. We now show a puzzle and its corresponding CNF formula. Note that all clauses added in the second or third component have length 2.

Consider the following Kamaji puzzle:

```
1 2 1
2 3 2
1 2 1
```

For this puzzle, the translation procedure generates the following list of clauses that have to be satisfied:

1. $(x_{001} \vee x_{002})$
2. $(x_{011} \vee x_{013})$
3. $(\neg x_{011} \vee \neg x_{013})$
4. $(x_{023} \vee x_{024})$
5. $(x_{102} \vee x_{105})$
6. $(\neg x_{102} \vee \neg x_{105})$
7. $(x_{124} \vee x_{126})$
8. $(\neg x_{124} \vee \neg x_{126})$
9. $(x_{205} \vee x_{207})$
10. $(x_{217} \vee x_{218})$
11. $(\neg x_{217} \vee \neg x_{218})$
12. $(x_{226} \vee x_{228})$
13. $(x_{001} \vee \neg x_{011})$
14. $(x_{011} \vee \neg x_{001})$
15. $(x_{002} \vee \neg x_{102})$
16. $(x_{102} \vee \neg x_{002})$
17. $(x_{013} \vee \neg x_{023})$
18. $(x_{023} \vee \neg x_{013})$
19. $(x_{024} \vee \neg x_{124})$
20. $(x_{124} \vee \neg x_{024})$
21. $(x_{105} \vee \neg x_{205})$
22. $(x_{205} \vee \neg x_{105})$
23. $(x_{126} \vee \neg x_{226})$
24. $(x_{226} \vee \neg x_{126})$
25. $(x_{207} \vee \neg x_{217})$
26. $(x_{217} \vee \neg x_{207})$
27. $(x_{218} \vee \neg x_{228})$
28. $(x_{228} \vee \neg x_{218})$

This formula is also an instance of 2-SAT. Instances of 2-SAT are instances of SAT, where each clause of the CNF contains exactly two literals. This is the case for this example, because each piece covers exactly two entries and each entry is covered by exactly two pieces. Not every CNF that corresponds to a Kamaji puzzle is an instance of 2-SAT. However, it is often the case that the majority of the clauses have length two.

4 SAT-solvers and MiniSAT

We have a function that takes a given Kamaji puzzle as input and converts it into a set of clauses that correspond to a formula in conjunctive normal form. The translation procedure that was described earlier yields a reduction of a Kamaji puzzle to an instance of SAT such that a solution to the puzzle corresponds to a satisfying assignment for the formula. After we have translated a Kamaji puzzle into a set of clauses, corresponding with a logic formula in CNF, we will use a SAT-solver to find a satisfying assignment. A SAT-solver is a program that takes an instance of SAT as input and seeks an assignment of its variables such that the entire formula holds true. It is possible to write ones own SAT-solver in a suitable programming language, but this is very time-consuming and not relevant for the purposes of this research project. We have thus chosen to make use of the open-source SAT-solver named MiniSAT [ES03]. In the sequel we give some general background on SAT-solvers.

We will first describe the backtracking-based algorithm for solving SAT which forms the core of all algorithms that solve instances of SAT. We will then discuss unit clause propagation and pure literal elimination (two techniques that help to improve the efficiency of SAT-solvers), and then we will shortly discuss two algorithms that were designed to solve instances of SAT. We will finish this section with a brief desription of MiniSAT.

4.1 Backtracking

One of the most straightforward ways to find a satisfying assignment of variables in a CNF formula is backtracking-based. It is the core of most of the state-of-the-art SAT-solvers of today and can be described as follows:

Step 1:
Consider the variables contained by the formula in some order.

Step 2:
Assign the value true to the next variable in the given order and add this assignment to the list of currently made assignments.

Step 3:
Remove all clauses that hold true as a result of the assignment of the variable made last (in Step 2 or Step 7), and discard all literals corresponding to the variable that holds false as a result of the assignment.

Step 4:
If there are no clauses left in the remaining formula, return true (a solution is found, see the list of currently made variable assignments). Otherwise, proceed to Step 5.

Step 5:
Does the remaining formula contain any empty clauses? That is, clauses that contain no literals. If so, proceed to the next step, else repeat the process from Step 2.

Step 6:
The current assignment of variables cannot lead to a solution. If the value true was assigned to the last variable we considered, proceed to Step 7. Otherwise, proceed to Step 8.

Step 7:
Assign false to the variable that was assigned last, make sure this change is reflected in the list of currently made variable assignments and return the structure of the formula to how it was before the value true was assigned to this variable in Step 2. Repeat from Step 3.

Step 8:
We have tried both true and false without success. Remove the last made variable assignment and go back to the preceding variable. Proceed to Step 6.

Various observations regarding the solvability of a formula in conjunctive normal form can be made based on its structure. In the past sixty years, this has led to the development of some additional rules that can be integrated into the operation of SAT-solvers to yield better performance. SAT-solvers utilize different techniques to seek solutions as efficiently as possible. They can be incredibly efficient, which is what makes it appealing to reduce problems to instances of SAT to enable their use. We will now discuss the strategies that SAT-solvers use to search for an appropriate assignment of the variables.

4.2 Unit Clause Rule

One well-known strategy that SAT-solvers utilize is called *Unit Propagation*. Unit Propagation occurs if there is a clause that contains only a single literal. Because the clause must evaluate to true in order for the entire formula to be true, the literal must hold true. If this scenario occurs, the literal is assigned the value true and the clause can be safely removed from the formula and so can all other clauses that contain the same literal (these clauses must all be true, because they contain a literal that is true). Besides that, all occurrences of the negative literal can be removed from the CNF since they are obviously false in any satisfying assignment of the variables. For example, if a CNF contains the single literal clause (x_1), then x_1 must be true and therefore all clauses containing x_1 are automatically true. Also, in that case, $\neg x_1$ is false and thus this literal can be removed from any clause that contains it. Unit Propagation will repeat this process until no more unit clauses remain.

4.3 Pure Literal Elimination Rule

Pure literal elimination occurs if one of the variables in a logic formula in conjunctive normal form is pure. A variable is said to be *pure* if either its corresponding negative or positive literal occurs in the formula, but not both. Suppose for example that $\neg x_{12}$ occurs in a CNF formula, but x_{12} does not. It is easy to find an assignment for the variable x_{12} such that all clauses that contain $\neg x_{12}$ are true. The SAT-solver can then make this assignment and eliminate all the clauses that contain the literal $\neg x_{12}$. Because x_{12} does not occur anywhere in the formula, this will have no effect on the solvability of the remaining set of clauses. For further reading on pure literal elimination we recommend [Joh05].

4.4 DPLL

DPLL is short for "Davis-Putnam-Logemann-Loveland" [DLL62]. This algorithm is effectively an extension of the backtracking-based algorithm for seeking a satisfying assignment. The 'extension' encompasses the integration of both Unit Propagation and pure literal elimination that were discussed earlier and lead to enhanced performance.

4.5 CDCL

CDCL is short for "Conflict driven clause learning" [SS96]. This algorithm for solving instances of SAT is inspired by the DPLL algorithm. The term "conflict driven"in the name of the algorithm reflects the fact that this algorithm involves the construction of an implication graph during the process of assigning values to the variables. If a conflict occurs, partial assignments leading to this conflict are cut off. A conflict in the implication graph will occur if an assignment leads to a contradiction in the implication graph. If this happens, the partial assignment (of the involved variables) will be cut off from the search space. For example, suppose that the value true is assigned to variables x_{10} and x_{21}, false is assigned to variable x_{15} and suppose that the implication graph conveys that this will lead to a contradiction. CDCL will then cut off the branch of the search space that

contains the partial assignment that led to the contradiction and will add a new clause that contains the negation of the partial assignment that led to the conflict, in this case $(\neg x_{10} \vee \neg x_{21} \vee x_{15})$. Integration of the principle of Conflict driven clause learning generally results in a substantial improvement in the overall performance of the algorithm. For further reading on the latest developments regarding SAT-solvers we recommend [LXL+18]

4.6 MiniSAT and its Inner Workings

MiniSAT encompasses many of the strategies for solving instances of SAT that were described in the previous subsection. DPLL and CDCL are still prominent in modern-day research on the Boolean satisfiability problem as they have played a central role in the early development stages of SAT-solvers and still form the core of many state-of-the-art SAT-solvers today. MiniSAT is no different in this regard and builds heavily on the principle of Conflict driven clause learning.

The MiniSAT program allows for the specification of an output file. If an output file is specified by the user, this file will contain the final result after MiniSAT has finished its operation. The output file will either state the word 'SAT' and list the values of the satisfying assignment it has found or it will simply state 'UNSAT', which indicates that no satisfying assignment was found. MiniSAT will find a solution if there is one, but it will never list all of the existing solutions. If one runs MiniSAT several times with the same CNF as input, it will always find the same solution, and unlike the Brute Force Search approach, MiniSAT invariably ends the search for a solution once it has found one. Nevertheless, it is possible to find all solutions by repeatedly adding a clause corresponding to the solution found by MiniSAT and running MiniSAT on the new formula.

5 Making Puzzles

There are many strategies one can think of, when it comes to making Kamaji puzzles. One of our aspirations in the early stages of this research project was to construct a deterministic algorithm that can generate a uniquely-solvable Kamaji puzzle for a given board size. Unfortunately, this turned out to be more difficult than we had anticipated. The construction of an algorithm that can create a single n by n puzzle with $n > 15$ in a short period of time, has proven to be a daunting task. What seems to be even more challenging is creating uniquely-solvable puzzles. Most puzzles that the puzzle-generator has produced contain many pieces of length 2, where one entry contains 1 and the other contains $m - 1$, where m denotes the "Maximum Value", leading to situations with multiple solutions. Still we have managed to write a computer program that can create Kamaji puzzles of size up to 18. We will first describe the operation of the puzzle-generator that we have implemented. In Figure 5, at the end of this section, we present two puzzles that were both created with this puzzle-generator. One has multiple different solutions and the other one is uniquely-solvable.

5.1 The Puzzle-Generator Program

Given an integer $n \geq 3$ that is provided as input, the puzzle-generator that we have implemented will attempt to create a valid n by n Kamaji puzzle. In our implementation, the puzzle's "Maximum Value" m is automatically set to the value of n. First, the puzzle-generator randomly selects an entry of the puzzle board that will contain m. Next, we fill the board with a number of pieces, all of which have length two and contain the numbers 1 and $m-1$. From here on, the algorithm will "randomly" place pieces onto the board. Here, by "randomly" we mean that the puzzle-generator selects a random entry, a random piece-length ℓ such that $\ell \leq (\frac{2}{3} \times n)$ and a random direction (right, bottom-right, down, bottom-left, left, top-left, above and top-right) in which the piece will be oriented, starting from the random entry that was selected first. However, we have reduced the probability that 1 or $m-1$ are selected to prevent the board from being dominated by these numbers. For example, entry $(2, 2)$, piece-length 3 and direction right may be selected by the program. This means that the next piece will cover 3 entries: (2,2), (2,3) and (2,4). Of course, the piece will be placed only if it does not exceed the boundaries of the puzzle board and does not have an overlapping entry that contains a number > 1 with some other piece. It will repeat the process of placing pieces onto the board, until either all entries of the board have been covered or it has failed to place a new piece for a given configuration of the board. If the first scenario occurs, a valid Kamaji puzzle has been generated and the set of pieces that have been placed form a solution to this puzzle, see Figure 5 for two examples. If the latter scenario emerges, the algorithm will remove the piece that was placed last and will continue from here. To avoid excessive runtimes of the program, in case a certain number of removals has occurred, the program will abort. In such event, the program has failed in its attempt to generate a puzzle.

2	5	1	1	1	4	3	2	1	1
8	1	4	9	4	1	9	8	9	2
8	6	2	1	1	3	1	6	4	1
7	4	6	3	7	1	3	2	1	3
3	2	3	5	8	1	4	2	1	3
2	8	2	1	1	8	1	6	1	4
3	4	6	9	4	5	4	1	7	9
5	9	1	8	5	6	2	1	7	3
8	4	9	2	9	9	9	5	4	4
5	2	2	3	1	4	10	3	6	6

10×10 puzzle, 240 solutions.

5	2	4	6	2	1	1
3	3	6	1	5	1	1
1	4	3	2	1	5	2
6	6	2	3	1	5	3
5	5	2	3	1	4	7
6	2	6	3	2	3	1
1	1	6	4	4	6	6

7×7, uniquely-solvable puzzle.

Figure 5: Left: 10×10 puzzle with 240 solutions. Right: 7×7 puzzle with 1 solution.

6 Experiments

We have conducted various experiments and in this section we will discuss them. First, we will shed some light on how frequently each integer value occurs on average in a Kamaji puzzle, given the puzzle's size and difficulty. Then, we will discuss the performance of *Biggerfirst* for puzzles of various sizes and difficulty. After that, we will compare the execution times of *Biggerfirst* with that of *Brute Force Search*. Next, we will discuss the performance of the puzzle-generator that we have implemented. More specifically, we will see how often the puzzle-generator manages to create a uniquely-solvable puzzle and how often it succeeds in creating a valid Kamaji puzzle at all. Finally, we briefly discuss usage of the SAT-solver.

We have used a set of puzzles that was distributed by "Denksport" [Den07]. This set consists of two 1-Star 7×7 puzzles, four 2-Star 7×7 puzzles, four 3-Star 7×7 puzzles, two 1-Star 8×8 puzzles, four 2-Star 8×8 puzzles, four 3-Star 8×8 puzzles, two 1-Star 9×9 puzzles, six 2-Star 9×9 puzzles and six 3-Star 9×9 puzzles. For the second experiment, in addition to puzzles from this set, we have used two 3-Star 10×10 puzzles and three 3-Star 11×11 puzzles. These were the largest puzzles we could find.

6.1 Frequency of Integer Values

"Denksport" utilizes a system of stars to indicate the difficulty of a given puzzle. There are three levels of difficulty. The easiest puzzles are marked by a single star. The most difficult puzzles are marked by three stars and moderate difficulty is indicated by two stars.

We have used puzzles of size 7×7, 8×8 and 9×9, as those are the only puzzle sizes of which, for each level of difficulty, we have at least two puzzles that have that difficulty. Table 1 shows how often numbers occur in these puzzles (NA means not applicable).

Value	**1-Star**			**2-Star**			**3-Star**		
	7×7	8×8	9×9	7×7	8×8	9×9	7×7	8×8	9×9
1	4.000	4.500	6.500	5.750	6.750	8.000	12.250	11.750	17.000
2	10.500	10.500	11.500	14.500	18.000	17.125	13.250	19.750	19.400
3	8.500	8.000	10.500	12.000	12.000	14.875	7.500	11.000	15.200
4	8.500	14.000	9.500	6.250	10.750	12.625	6.250	7.000	7.800
5	10.500	8.000	9.500	6.500	5.750	9.750	6.000	5.500	9.000
6	6.000	10.500	10.500	3.000	5.750	7.750	2.750	5.750	4.400
7	1.000	7.500	11.500	1.000	4.000	5.500	1.000	2.250	3.800
8	NA	1.000	10.500	NA	1.000	4.250	NA	1.000	3.400
9	NA	NA	1.000	NA	NA	1.000	NA	NA	1.000

Table 1: Average number of occurrences per integer value in puzzles, depending on size/difficulty.

Note that the average frequency of occurrence of the integer n for a puzzle of size $n \times n$ is always 1. This is because the Maximum Value m is equal to n for all puzzles that we used.

It is important to observe that puzzles of higher difficulty contain relatively more small than large integers. Earlier in this thesis we have stated that The larger the integer value that is contained by an entry, the larger the probability that only one piece exists that can cover this entry. The reverse is also true. That is, the smaller the integer contained by an entry, The larger the probability that multiple pieces exist that cover this entry. The results shown in Table 2, in the next subsection, confirm this. Also, if the puzzle board contains more small integers, the pieces that make up the solution have greater average length. All this means that it is generally harder for one to solve the puzzle, which is exactly what you would expect for puzzles that belong to a category of "more difficult" puzzles.

6.2 Biggerfirst Experimentation

We have tested the performance of *Biggerfirst*. We have used the same set of puzzles we discussed in the previous subsection.

In this experiment we have measured, for given sets of puzzles of equal size and difficulty, how many runs *Biggerfirst* requires to solve each puzzle and how many pieces it adds to the partial solution for each of the runs.

	1-Star			**2-Star**			**3-Star**		
	7×7	8×8	9×9	7×7	8×8	9×9	7×7	8×8	9×9
Average Run Count	2.00	1.50	2.50	2.50	3.00	3.20	3.25	3.25	3.40
Found in 1^{st} Run	86	96	94	80	76	72	61	54	61
Found in 2^{nd} Run	14	4	5	13	21	19	25	30	26
Found in 3^{rd} Run	NA	NA	1	7	1	8	11	15	10
Found in 4^{th} Run	NA	NA	NA	NA	1	1	3	1	3
Found in 5^{th} Run	NA	NA	NA	NA	1	NA	NA	NA	NA

Table 2: Various measurements of the performance of *Biggerfirst*.

The row labeled "Average Run Count" shows the average number of runs needed to solve a puzzle of a given size and difficulty. The other rows show what percentage of the sum of the number of pieces that were found for each puzzle of a given size and difficulty, were found in a respective run. For instance, the leftmost entry of the row labeled "Found in 1^{st} Run"contains 86, indicating that 86% of the pieces were found in the first run.

Based on Table 2 we observe the following:

1. *Biggerfirst* succeeds in solving all puzzles for the given set. The application of Brute Force is not needed.

2. It is consistently the case that, for a set of puzzles of size $n \times n$, as the level of difficulty increases, so too does the average number of runs that *Biggerfirst* requires to solve these puzzles.

3. It is also consistently true that *Biggerfirst* will find relatively fewer pieces in the first run of its operation, as the difficulty increases for a puzzle size $n \times n$.

6.3 Runtime Comparison: BruteForce vs Biggerfirst

For this experiment, we have measured for the afore-mentioned set of puzzles of sizes ranging from 7×7 up to and including 11×11, the time it takes to solve the puzzle 500,000 times. We have taken these measurements for both *BruteForce* and *Biggerfirst*. Figure 6 displays the average runtime for puzzles of various sizes, all of which are marked by three stars (highest level of difficulty).

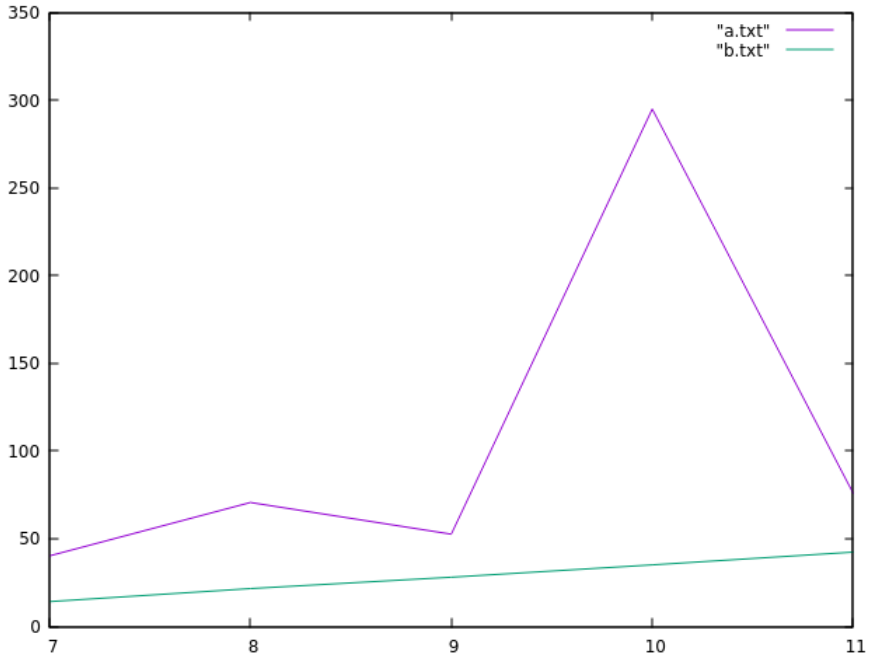


Figure 6: Execution time in seconds for 500,000 runs of *Biggerfirst* (green line) and 500,000 runs of BruteForce Search (purple line).

Figure 6 shows that the average runtime increases for both algorithms as the puzzle size goes up, which obviously makes sense. The graph of *Biggerfirst* (represented by the blue line) is quite resemblant of a straight line, whereas the slope of the graph of BruteForce (represented by the red line) is much less stable. This is due to the fact that BruteForce tries to cover the entries in a specific order, whereas *Biggerfirst* jumps from one location to the next, because it considers the entries in an order that is based on their relative size.

6.4 Puzzle Creation Experiments

We have run experiments that involve the puzzle-generator program that was discussed in Section 5 of this thesis. Our primary motivation for creating puzzles was the scarcity of available Kamaji Puzzles, especially those of larger size. As mentioned earlier, it is very challenging to create large puzzles that are uniquely-sovable. Nonetheless, we wanted to have some measurement regarding the performance of our puzzle-generator. Hence, we have come up with this experiment, which we will now describe.

For this experiment, we have created puzzles of sizes 4×4, 5×5, 6×6 and 7×7. For each of these puzzle sizes, we have made 1000 attempts to create a puzzle. Here, by an attempt we mean a single run of the program, which either succeeds (yields a solvable puzzle) or fails (fails to construct one). The results of the experiment can be found in Table 3.

	4×4	5×5	6×6	7×7
Total Attempts	1000	1000	1000	1000
Failed Attempts	55	225	320	492
Puzzles Created	945	775	680	508
Uniquely Solvable	98	83	65	76

Table 3: Performance of the puzzle-generator.

One obvious observation is that the number of failed attempts increases as the puzzle size goes up, but for the number of uniquely-solvable ones seems to stabilize.

6.5 Using the SAT-Solver

We have used the translation procedure described in Section 3 to produce CNF formulas corresponding to the Denksport puzzles. We have applied MiniSAT (see Section 4) to these formulas, and all of them were solved almost instantaneously. One property of these formulas is that most of their clauses contain only two literals, which is part of why MiniSAT solves them so quickly.

7 Framework and Implementation

We have implemented the framework along with all the strategies and the ability to input Kamaji puzzles in C++, using the integrated development environment Codeblocks. [Cod] The resulting C++ program can read and process Kamaji puzzle boards stored in input files in simple text format. In order to use the Kamaji that is displayed in Figure 1 as input we simply create a text file with the following lines of characters:

```
4
5
2 1 4 1
2 3 5 4
2 1 3 1
1 4 2 3
```

The first line contains the size of the board along the two dimensions. In this case, the board is 4 by 4 in size, thus the number on the first line of the input file is the number 4. The second line contains the Maximum Value which is 5. The following lines represent the structure of the board. We have chosen to specify each subsequent row on a separate line as this yields an intuitive and programmatically convenient representation of the board. The framework will process the input puzzle by storing it in a two-dimensional array of type integer, uses a second two-dimensional integer type array to construct a solution to the puzzle and a third one to store the solution. Some Kamaji puzzles have multiple solutions. If *Biggerfirst* or one of the Brute Force Search variants is applied, only the solution that is found first will be stored, but the program may display all the other solutions that it finds and it will inform the user whether any solution exists and if so, whether the solution is unique or not.

The Functionality of the main program The program first asks the user which file he or she would like to provide as input. If the file does not exist or cannot be opened due to some other issue, the program will immediately write an error message to the terminal and abort the program altogether. The input file must represent a Kamaji puzzle and it should be structured like the sample input file displayed in Figure 1. It is neccessary to provide the dimension size and "Maximum Value"of the puzzle board on the first two lines in the stated order, otherwise the program will not function properly. If all goes well, the program will provide the user with a number of options, one for each of the solution search strategies that were discussed in this thesis. That way, the user can select how he or she would like the puzzle to be solved. The program will then go on to solve the puzzle by application of the solution search strategy that was selected, except if the reduction to SAT was chosen, in which case the program will only create a text file that contains the puzzle's CNF translation in DIMACS/CNF format and two other files that are required for the back-translation of solutions produced by SAT.

8 Conclusions and Future Work

In this thesis we examined Kamaji puzzles. In these puzzles, squares of integers, we have to find consecutive series of numbers that sum to a given value.

There exist many different strategies to solve Kamaji puzzles that can be easily implemented and are quite fast. It would be interesting to implement some more of these strategies in future work to make further comparisons. Especially, a strategy that comprises the utilization of a neural network, because such a strategy could closely resemble the way humans solve Kamaji puzzles. The greatest challenge however still remains the creation of puzzles of a large size and puzzles that are uniquely solvable. Is there a deterministic algorithm for creating puzzles uniquely solvable puzzles? These issues can be adressed in future research into the Kamaji puzzle.

8.1 Future Work

Not much research has been conducted into Kamaji puzzles yet to our knowledge, and Kamaji puzzles are no longer being sold. Nevertheless, the Kamaji puzzle is very susceptible for future work, especially with regards to the making of Kamaji puzzles. We will now discuss some potential topics for future research into Kamaji's.

8.1.1 Solution Search Strategies

The solution search strategy *Biggerfirst* that we have implemented attempts to find a valid combination for the entries that contain the largest, unused numbers. This strategy is based on the observation that the probability that only a unique valid combination exists for a given entry is greater for larger numbers. The same logic applies to entries that have fewer available neighbours. Where *Biggerfirst* orders the entries of a puzzle by the size of the integers that they contain, a new solution search strategy can be implemented that orders the entries by their number of available neighbours. We have seen that *Biggerfirst* yields substantially shorter execution time for puzzles with a unique solution than Brute Force Search does. It would be worthwhile to implement this strategy in future work and compare its efficieny to that of other solution search strategies.

Another alternative is to solve Kamaji puzzles with the aid of a neural network. In order to effectively do so, one could, for example, feed a neural network with images of Kamaji puzzles and their solutions represented similarly to how this is done in Figure 1. One interesting fact about such a strategy is that it is more resembling of how a human solves the puzzle considering the puzzle as a whole. All of the afore-discussed strategies can only consider puzzle entries one by one. All these strategies process a puzzle in the form of a two-dimensional array, where the array's indices represent the puzzle's entries. A human being can literally see the whole picture and perhaps a computer program can do the same if a neural network is used as described here.

8.1.2 Puzzle Creation Strategies

A very interesting area of research with regards to Kamaji puzzles concerns the creation of Kamaji puzzles. One can think of countless strategies when it comes to the making of puzzles. The puzzle creation strategy that we discussed in this thesis starts with an empty board and randomly places pieces on the puzzle board. We will discuss some more strategies that can be implemented and analysed in future work.

The puzzle creation strategy that we discussed in this thesis starts off with an empty board and randomly places pieces on the puzzle board. If by doing so, the program manages to cover each puzzle entry, a puzzle has been created successfully. As the size of a to be constructed puzzle increases, it becomes more challenging to cover all of the entries using this strategy. Thus, one strategy that can be implemented in future research goes as follows. Given is an empty puzzle board that is n by n in size. Divide the board into smaller non-overlapping square-shaped and/or rectangle-shaped boards. Write a program that, for each of these smaller boards, applies the algorithm for creating puzzles that was described in this thesis. If the program succeeds in filling up each of these smaller boards with pieces such that all their entries have been covered by some piece, putting all these smaller boards back together in the configuration of the original n by n puzzle board yields a Kamaji puzzle.

Another alternative strategy for creating Kamaji puzzles, also a modification of the strategy described in this thesis, fills an empty puzzle board with pieces that are empty. That is, pieces of which only the entries that they cover are specified. Because the integers contained by the entries of a piece are not specified immediately after the piece is placed, it is straightforward how one can fill the values once all entries have been covered by at least one piece. The process for doing so is more likely to yield a puzzle than the strategy that we implemented for this thesis. However, there is one pitfall to the placement of the pieces. If all entries of a piece are overlapping entries, that means they all contain 1 and will add up to the number of entries the piece contains. Therefore, each piece that has a length ℓ such that $\ell < m$, where m denotes the "Maximum Value"of the puzzle, must have at least one non-overlapping entry to assure that its entries can eventually sum up to m.

References

[Ber81] H. J. Berliner. An examination of brute force intelligence. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence, IJCAI '81, 1981*, pages 581–587, 1981.

[Bia12] M. De Biasi. Binary puzzle is NP-complete, 2012.

[Cod] CodeBlocks. CodeBlocks IDE. http://www.codeblocks.org [Online; Accessed 15-december-2017].

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Den07] Denksport. Kamaji puzzels, circa 2007.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In *6th International Conference onTheory and Applications of Satisfiability Testing, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[Fre] Freudenthal instituut. http://www.fi.uu.nl [Online; Accessed 05-april-2018].

[Joh05] J. Johannsen. The complexity of pure literal elimination. *J. Autom. Reasoning*, 35:89–95, 2005.

[LXL+18] C. Li, F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li. Clause vivification by unit propagation in CDCL SAT solvers. *CoRR*, abs/1807.11061, 2018.

[OL06] J. Ouaknine and I. Lynce. Sudoku as a SAT problem. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2006*, 2006.

[SS96] J. P. Marques Silva and K. A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *Eigth International Conference on Tools with Artificial Intelligence, ICTAI '96*, pages 467–469, 1996.