School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCE

Killer Sudoku as a Constraint Satisfaction Problem

Henry Davies

Submitted in accordance with the requirements for the degree of Computer Science BSc

2019-2020

40 credits

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Source code	GitLab repository: `https://gitlab.com/sc16hd/sc16hd_project`	Public (08/05/2020)
Written report	PDF	Minerva (08/05/2020)

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) Henry Davies

Summary

A constraint satisfaction problem (CSP) is way of formulating a mathematical problem. CSPs are studied in the field of artificial intelligence. Killer Sudoku is a popular variant of the famous Sudoku puzzle. This project expresses Killer Sudoku as a CSP and compares the performance of different computational methods for solving it. Solving is performed by software built as part of the project that implements constraint solving methods.

Acknowledgements

I would like to thank my supervisor Dr. Isolde Adler for her guidance and support throughout the project. I would also like to thank my assessor Dr. Brandon Bennett for providing useful feedback.

Contents

Chapter 1

Introduction

1.1 Overview

A constraint satisfaction problem (CSP) is way of formulating a mathematical problem. A CSP can be solved using constraint satisfaction methods. There has been much study into improving constraint satisfaction, both by increasing the efficiency of existing methods and creating new methods. Improving constraint satisfaction methods is important because they can be used to solve real life problems, such as scheduling and resource allocation [1]. Constraint satisfaction methods can be developed by observing their performance on theoretical problems, such as the 4 Queens problem and map colouring problems.

On 12 November 2004 *The Times* newspaper began publishing Sudoku puzzles in the puzzle section [2]. The puzzle presents a seemingly simple challenge: fill the puzzle grid with the numbers 1 to 9 such that no row, column or box contains the same number. This challenge soon began to captivate the minds of thousands, and the puzzle's popularity boomed. Following up on the popularity of Sudoku variations began to enter circulation that added additional rules, presenting fresh challenges to solvers. These include Killer Sudoku, Kakuro and Jigsaw Sudoku.

Sudoku puzzles have been the subject of much study in the field of constraint satisfaction. However, the study of its variants, namely Killer Sudoku, has been minimal in comparison. This project will study Killer Sudoku as a CSP.

1.2 Project aims and objectives

The aim of the project is to express Killer Sudoku as a CSP and compare the performance of different constraint satisfaction methods for solving it. The objectives of the project are:

- Introduce the area of constraint satisfaction
- Introduce Killer Sudoku and express it as a CSP
- Build a solver than solves Killer Sudoku puzzles using a variety of constraint satisfaction methods
- Solve Killer Sudoku puzzles using a general purpose constraint solver
- Analyse and explain the performance of different solving methods

1.3 Deliverables

The deliverables for this project are the following:

- Source code for the solver. This is stored in a GitLab repository found here: `https://gitlab.com/sc16hd/sc16hd_project`. A README file is included with installation instructions.
- The final report. This includes details on operating the solver.

Chapter 2

Project methodology

2.1 Initial plan

Work on this project began in October 2019. This early work focused on selecting a CSP to study and carrying out background research. This work was not particularly structured and did not follow a plan. Development of the solver began in January 2020. The work was organised into the following segments:

1. Develop solver implementing human techniques
2. Develop solver implementing search techniques
3. Solve Killer Sudoku puzzles using general purpose constraint solvers
4. Develop a tool to speed up translation of Killer Sudoku puzzles into a computer readable format
5. Conduct a comparison of the performance of different solving methods for solving Killer Sudoku puzzles
6. Write the final report

The development work required for segments 1, 2, 3 and 4 could be performed independently, so the agile methodology was adopted. This allowed any major difficulties in developing each segment to be discovered early on. It also allowed integration of each segment at early stages of development. This mitigated the risk of integration becoming a huge task. This risk typically arises when following the waterfall methodology.

2.2 Risk mitigation

There were no external dependencies for this project. Therefore the only risks for the project were facing unexpected difficulties that could have had significant impacts on development, and not being able to devote as much time to the project as intended. These risks could be mitigated by decreasing the complexity of the project. This could be achieved by taking some/all of the following actions:

- Implement fewer human methods than originally intended
- Implement fewer search methods than originally intended
- Do not integrate human and search methods
- Do not develop a tool to speed up translation of Killer Sudoku puzzles into a computer readable format
- Compare the performance of solving methods less extensively

Chapter 3

Background Research

3.1 What is a constraint satisfaction problem?

A constraint satisfaction problem (CSP) is way of formulating a mathematical problem. A CSP is defined by a set of variables, domains and constraints and is solved by applying constraint satisfaction methods [3]. These methods can be split into two types, constraint propagation and search.

3.1.1 Formal definition

A CSP is formally defined by a triple (X, D, C) [3].

X is a set of variables: $\{x_1, ..., x_n\}$
D is a set of domains: $\{D_1, ..., D_n\}$
C is a set of constraints: $\{C_1, ..., C_m\}$

Each variable x_i has an associated domain D_i. A variable x_i can be assigned to any of the values in D_i. A constraint defines the combination of values that a subset of variables can be assigned. A constraint C_j is defined by a tuple, (t_j, R_j). $t_j \subset X$ is a subset of k variables and R_j is a k-ary relation on the corresponding subset of domains [3]. A constraint C_j is satisfied if the values assigned to variables t_j satisfy the relation R_j.

An *assignment* is a mapping of variables to values. An assignment is *consistent* if all constraints are satisfied. An assignment is *complete* if all variables are assigned. An assignment that has any unassigned variables is a *partial* assignment. An assignment is a *solution* if it is both *consistent* and *complete* [4].

We'll now introduce a very simple CSP in words, and then express it in the form (X, D, C):

'x must be smaller than y. x can take the value of 2, 5 or 7. y can take the value of 3, 4 or 6.'

There are only two variables, x and y:

$$X = \{x, y\}$$

The domain of x is 2, 5 and 7, and the domain of y is 3, 4 and 6:

$$D_x = \{2, 5, 7\}$$
$$D_y = \{3, 4, 6\}$$
$$D = \{\ \{2, 5, 7\}\ ,\ \{3, 4, 6\}\ \}$$

There is only one constraint, which is that x must be smaller than y. This is a *binary* constraint, as it relates exactly two variables. Labelling the constraint with the number 1, the constraint is:

$$C_1 = (\ \{x, y\}\ ,\ \{(2,3),\ (2,4),\ (2,6),\ (5,6)\}\ )$$

As this is the only constraint, the set of constraints C is:

$$C = \{\ (\ \{x, y\}\ ,\ \{(2,3),\ (2,4),\ (2,6),\ (5,6)\}\ )\ \}$$

3.2 Constraint propagation

Constraint propagation is a term used to describe any reasoning that updates the existing set of domains or set of constraints to be more *restrictive* [5, p. 5]. A domain is made more restrictive by removing one or more of its values. This is called *filtering*. The set of constraints are made more restrictive by modifying existing specific constraints to permit fewer assignments or by adding new constraints. Constraint propagation is achieved via inference on the existing domains and constraints. When a CSP is made more restrictive, an *equivalent* CSP is produced. Equivalent CSPs have the same set of solutions [4]. Due to the more restrictive constraints and domains the equivalent CSP is simpler and easier to solve.

Methods of constraint propagation can be organised into two different categories, enforcing *local consistency* conditions and iteration of *reduction rules*.

3.2.1 Enforcing local consistency conditions

Enforcing a local consistency condition filters the domains of a subset of variables such that all possible assignments of the variables satisfy a subset of constraints. The subset of constraints is determined by the level of local consistency that is being enforced. There are several levels of local consistency, increasing from node consistency to arc consistency to path consistency. The higher the level of consistency, the more restrictive the CSP becomes. The computational cost of enforcing a level of local consistency increases as the level increases however. There exist several different algorithms for enforcing each level of consistency.

3.2.1.1 Node consistency

A variable x_i is node consistent if for all possible assignments of x_i to values in D_i, every unary constraint on variable x_i is satisfied [4]. A unary constraint concerns only one variable. Local node consistency can be enforced by removing all elements of D_i that do not satisfy the unary constraints on variable x_i. For example, a variable x_1 has domain $D_1 = \{1, 2, 3, 4, 5\}$. A unary constraint C_1 constrains x_1 to be greater than 2. Enforcing node consistency removes values from the domain that do not satisfy the constraint, resulting in D_1 being filtered and updated to $\{3, 4, 5\}$.

A CSP is globally node consistent if all variables are node consistent. Global node consistency can be achieved by enforcing local node consistency on all variables. As unary constraints only

concern a single variable, global node consistency can be enforced in one pass [4].

3.2.1.2 Arc consistency

A variable x_i is arc consistent with variable x_j if for every value $a \in D_i$ there exists a value $b \in D_j$ such that the assignment of x_i to a and x_j to b satisfies the binary constraint between variables x_i and x_j [4]. If the assignment of variable x_i to a and x_j to b satisfies the binary constraints, then *a supports b*. An arc $arc(x_i, x_j)$ represents the binary constraint imposed on x_i by x_j. To enforce arc consistency on $arc(x_i, x_j)$, all values $a \in D_i$ that are not supported by a value $b \in D_j$ are removed from D_i. For example, look back at the CSP introduced in section 3.1.1. If arc consistency is enforced on $arc(x, y)$, 7 is removed from D_x, as there is no value in D_y that supports assigning x to 7. Arc consistency is directional, meaning that if x_i is arc consistent with x_j it does not necessarily mean that x_j is arc consistent with x_i.

This definition for arc consistency is limited to *binary normalised* CSPs. A binary normalised CSP contains binary and unary constraints only. It is possible to encode every non-binary constraint as a binary constraint [6].

If arc (x_i, x_j) is known to be consistent and arc consistency is enforced on arc (x_j, x_k), arc (x_i, x_j) is no longer guaranteed to be consistent. Therefore global arc consistency cannot be enforced in one pass. A simple algorithm developed for enforcing global arc consistency is AC-1. In a single 'pass' the the algorithm enforces arc consistency on every arc in turn. If any domains are filtered, the algorithm performs another pass. If a pass does not filter any domains the algorithm ends, indicating the CSP is globally arc consistent [4].

3.2.1.3 Higher level consistencies

Arc consistency is considered the most important level of local consistency [5, p. 91]. This is due to its use in solvers and because improvements in the efficiency of arc consistency algorithms can be applied to algorithms that enforce other levels of consistency [5, p. 40]. However, higher levels of consistency do exist such as path consistency. Although enforcing higher levels of consistency achieves more propagation than enforcing lower levels consistency, the computational overhead in doing so is large [4].

3.2.2 Rules iteration

A reduction rule specifies the conditions necessary for filtering of a domain (or domains) to occur [5, p. 68]. In section 3.1.1 constraints are formally defined with a relation that determines the assignments the constraint permits. However, constraints are rarely implemented in this way for two reasons. The first is that storing all permissible assignments for a constraint takes up a lot of space. The second is that most common constraints have a structure that can be taken advantage of when performing constraint propagation [5, p .497]. It is much harder to take advantage of this structure if the constraint is implemented using the set of permissible assignments. Implementing constraints using *propagators* addresses both these issues. A propagator, p, is a function that maps domains to domains. Constraints are effectively

implemented by a propagator or collection of propagators. Iterating propagators achieves constraint propagation, hence the name 'rules iteration'. Two properties of propagators are *decreasing* and *correctness* [5, p .497]:

- **Decreasing** - For all domains D, $p(D) \subseteq D$
- **Correctness** - p will only filter values that cannot exist in a solution

If a propagator is not decreasing the propagator could potentially add values to domains, achieving the opposite affect of constraint propagation. If a propagator is not correct, it may falsely indicate a solution does not exist. These two properties ensure that propagation will terminate and is correct. A propagator is at a *fixed point* if executing the propagator does not filter any values. A *mutual* fixed point is reached when all propagators are at a fixed point [7, p .23].

The *strength* of a propagator is determined by how much filtering the propagator performs. If propagators p_1 and p_2 implement the same constraint, p_1 is **stronger** than p_2 if for all domains D, $p_1(D) \subseteq p_2(D)$. Stronger propagators filter more values than weaker propagators do for the same constraint. As a result, using stronger propagators over weaker propagators makes the CSP more restrictive. However, stronger propagators usually have higher algorithmic complexity compared to weaker propagators, resulting in a higher cost of execution. Therefore it is important to balance the amount of filtering performed by a propagator and the propagator's run time when selecting a propagator to execute [7, p .21].

3.2.2.1 Propagator scheduling

To achieve efficient constraint propagation propagators must be executed according to a schedule. One method of scheduling is *event directed scheduling* [7, p .51]. An *event* describes a modification to a domain. Propagators *subscribe* to events. When a propagator is executed, it may trigger an event. Propagators that are subscribed to the triggered event are scheduled in a queue. Propagators are executed sequentially from the queue, possibly triggering further events leading to further propagators being scheduled. Once the queue is empty a mutual empty fixed point has been reached. It is then not possible for any propagator to perform any more filtering.

A naive way of implementing event directed scheduling is to firstly create an event for every domain that describes any modification to the domain. Every propagator subscribes to all events. This is clearly inefficient, as some propagators that are queued may have no dependence on the modified domain, and thus will perform no filtering when executed. A more sophisticated approach is to only subscribe propagators to events that describe modifications to domains that are relevant to the propagator.

3.2.3 Global Constraints

A *global constraint* implicitly defines the relation between a non-fixed number of variables [8] [5, p .169]. An example is the `alldifferent`$(x_i, ..., x_n)$ constraint. As the name suggests, this

constraint forces all variables $x_i, ..., x_n$ to be assigned different values. This is the formal definition of the constraint [9]:

$$\texttt{alldifferent}(x_1, ..., x_n) = \{\ (d_1, ..., d_n) \mid d_i \in D_i,\ d_i \neq d_j\ for\ i \neq j\ \}$$

It is possible to express every global constraint explicitly using a number of binary constraints. For example, the constraint $\texttt{alldifferent}(x_1, x_2, x_3)$ could be expressed instead by 3 binary constraints $x_1 \neq x_2$, $x_1 \neq x_2$ and $x_2 \neq x_3$. In this case only 3 explicit constraints are needed, but if a constraint that restricted 10 variables from having the same value was required, then 45 binary constraints would be required. It's obvious that modelling with global constraints has the advantage of making modelling simpler. Another advantage is that propagating a global constraint is more powerful than propagating all the equivalent binary constraints [10]. To show this lets create some domains for the variables x_1, x_2 and x_3 in this example: $D_1 = \{5, 7\}$, $D_2 = \{5, 7\}$ and $D_3 = \{5, 7, 8\}$. Propagating each binary constraint (equivalent to enforcing arc-consistency) would not result in any filtering. However, propagating the `alldifferent` constraint would filter values 5 and 7 from D_3.

3.3 Search

Search methods systematically explore the search space of a CSP attempting to find solutions. The search space of a CSP is the set of all possible assignments, both partial and complete. A search algorithm employs a strategy to explore the search space. A *complete* search algorithm (not to be confused with complete assignment) will always find a solution if one exists [5, p .85]. Therefore search algorithms can be used to show whether a solution exists or not for a given CSP.

3.3.1 Search algorithms

3.3.1.1 Generate and test

Generate and test is a very simple search algorithm. Every possible complete assignment is generated and tested to see if it is a solution. In the worst case every possible complete assignment will be generated before a solution is found. This is equal to the Cartesian product of all domains. Generate and test is very inefficient compared to other search algorithms [4].

3.3.1.2 Backtrack

Backtrack performs a depth-first search that incrementally builds a solution by extending a partial assignment [4]. Backtrack is a complete search. When backtrack is first called the partial assignment is the empty assignment (no variables have been assigned). Backtrack must be provided with a *variable order*. The variable order determines the order by which variables are 'processed'. The default variable order usually orders variables in the order they were defined [11]. When a variable is processed, backtrack attempts to extend the partial assignment by assigning the variable to a value in its domain such that the extended assignment is consistent. If every possible assignment of the variable to a value in its domain is inconsistent, the search 'backtracks'. This indicates that the partial assignment cannot be extended to a

solution, and is called a 'dead-end'. Backtrack unassigns the current variable, and then starts reprocessing the previous variable in the variable order.

Backtrack is highly susceptible to *thrashing*. Thrashing occurs when search fails repeatedly for the same reason [4]. To explain thrashing lets create a CSP:

Variables: x_1, x_2, x_3
Domains: $D_1 = \{4, 6, 7\}$, $D_2 = \{1, 3\}$, $D_3 = \{2, 6\}$
Constraints: $x_1 > x_2$, $x_1 = x_3$

Backtrack processes x_1 and x_2, assigning $x_1 = 4$ and $x_2 = 1$, as these values satisfy the constraint $x_1 > x_2$. Backtrack then processes x_3. Backtrack attempts to extend the partial assignment by assigning x_3 to 2 and 6 in turn. Neither of these assignments satisfy the constraint $x_1 = x_3$, so a dead-end has been reached. The search backtracks, and starts reprocessing x_2, this time assigning $x_2 = 3$. Backtrack then starts processing x_3. Once again, the partial assignment cannot be extended, as neither assignment of x_3 to 2 or 6 satisfies the constraint $x_1 = x_3$. Search has once again reached a dead-end. Backtrack 'backtracks', and starts reprocessing variable x_1 as D_2 has now been exhausted. In this case search has failed twice for the same reason, which is that the assignment $x_1 = 4$ cannot exist in a solution.

Thrashing is common with backtrack as the search does not consider which variable assignments are causing inconsistency.

3.3.1.3 Backjump

Backjump search was designed to address thrashing [4]. Backjump is a complete algorithm. Backjump also performs a depth-first search, but instead of performing a 'backtrack' performs a 'backjump'. Instead of reprocessing the previous variable in the variable order when unable to extend a partial assignment, backjump reprocesses the latest variable in the variable order whose assignment caused an inconsistency. In the CSP above, the partial assignment could not be extended to x_3 because the constraint $x_1 = x_3$ could not be satisfied. No constraint concerning variable x_2 was violated. Therefore the latest variable in the variable order whose assignment caused inconsistency was x_1. Backjump 'backjumps', and starts reprocessing x_1. Backjump skips the futile work that was performed by backtrack.

3.3.1.4 Forward checking

By integrating search and constraint propagation methods it is possible to create hybrid methods [5, p .89]. Forward checking is a hybrid method. Forward checking is a complete algorithm. Forward checking guarantees arc consistency between all assigned variables in the current partial assignment [5, p .63]. For a given variable x_i, forward checking extends the current assignment by assigning x_i to a value in its domain. A consistency check is then performed by enforcing arc consistency on every constraint concerning x_i. For example, in the above CSP when forward checking processes x_1 it assigns $x_1 = 4$. Arc consistency is then enforced on every constraint concerning x_1, which are the constraints $x_1 > x_2$ and $x_1 = x_3$.

Enforcing arc consistency on $x_1 > x_2$ results in no filtering of D_2. Enforcing arc consistency on $x_1 = x_3$ filters both 2 and 6 from D_3. D_3 is now empty. This indicates that the assignment of $x_1 = 4$ cannot exist in a solution, as there is no value that can be assigned to x_3 such that constraints are satisfied.

The advantage of forward checking compared to backtrack and backjump is that dead-ends are discovered earlier. In this case the assignment of $x_1 = 4$ was identified immediately as an assignment that could not exist in a solution. Backtrack and backjump had to process variable x_3 before this was realised. Discovering dead-ends earlier increases the efficiency of search, as less time is wasted attempting to extend partial assignments that cannot be built into solutions. The efficiency of forward checking therefore should be greater than backtrack and backjump. However, when forward checking assigns a variable there is an additional cost of enforcing arc consistency. If the total cost of enforcing arc consistency throughout the duration of search outweighs the saved cost of discovering dead-ends earlier, forward checking will not be more efficient than backtrack and backjump [12].

3.3.2 Variable ordering

The simplest variable ordering orders variables in the order they were defined. However, other variable orderings exist that can improve the efficiency of search. Orderings may be either *static* or *dynamic*:

- **Static** - The ordering is fixed before search begins and never changes
- **Dynamic** - A new ordering is produced after each variable is assigned

An ordering is produced by running an ordering heuristic on the unassigned variables. Backtrack and backjump can only work with static orderings [11]. Forward checking can work with dynamic orderings. A common scheme for variable ordering comes from the 'fail-first' principle, which says "To succeed, try first where you are most likely to fail" [13, p .95]. For search to find a solution it must assign all variables, which it does by extending a partial assignment. Search may encounter dead-ends when attempting to extend the assignment. It is better to discover dead-ends early on in search so partial assignments that do not lead to solutions are not explored. One way of realising the fail-first principle is implementing a heuristic that orders variables by the number of values in their domains in increasing order. Using this heuristic the next variable chosen is always the variable that has the fewest values in its domain. This heuristic performs very well on randomly generated problems but is not always the best choice [11].

Another heuristic that realises the fail-first principle is choosing the next variable to be the variable which has the most constraints with assigned variables [14]. This is because if a variable is concerned by more constraints than another variable is, assigning a value to the variable is likely to be harder.

3.3.3 Run time

The run time of each search algorithm depends on the number of dead-ends the search encounters before a solution is found. This in turn is dependent on the restrictiveness of the CSP. As we know, a CSP can be made more restrictive through constraint propagation. Therefore the more constraint propagation performed, the higher the increase in the efficiency of search [4]. Constraint propagation comes at computational cost. The cost of constraint propagation and search must therefore be balanced to achieve efficient solving. Constraint propagation can make a CSP restrictive such that a *backtrack-free* search is possible [4]. This means that when backtrack searches the CSP for a solution it finds a solution without discovering any dead-ends, and hence doesn't need to 'backtrack'.

3.4 Constraint Programming Systems

Constraint Programming Systems have been built to solve CSPs. We will refer to them as 'constraint solvers'. Constraint solvers are general purpose, meaning they are not built to solve specific problems. Constraint solvers perform constraint propagation and search techniques. Constraint propagation is implemented using propagators [5, p .497]. To solve a CSP using a constraint solver the CSP must firstly be modelled in a language the solver understands. This involves defining all variables, domains and constraints. It is essential for propagation based constraint solvers to implement efficient propagators for 'high level' constraints [7, p .9]. Global constraints are high level constraints, so it is important to use global constraints when modelling where possible. The propagators implemented by a constraint solver are sufficient for solving many problems but solvers can be extended by implementing additional propagators if desired [5, p .496].

3.4.1 MiniZinc

MiniZinc is a language used to model CSPs [15]. Using the MiniZinc program MiniZinc models can be compiled and solved by a range of constraint solvers. These constraint solvers include Gecode, Chuffed and Gurobi. The model does not have to specify how the solver should attempt to solve the problem (constraints to propagate or search method to use, for example). Annotations can be used to instruct the solver how to solve the problem if desired though. For example, the solver can be instructed to use fail-first ordering when searching. [15, p .69]. Modelling is very easy using the MiniZinc language compared to implementing a model directly using constraint solvers. MiniZinc has its own IDE and can be easily integrated into Python using the 'MiniZinc Python' package [16].

This project aims to compare different methods for solving Killer Sudoku. Although constraint solvers cannot be considered as a single 'method' for solving CSPs as they utilise a wide range of methods when solving, examining their performance is still relevant to this study. It is not relevant however to understand the intricacies of specific constraint solvers.

3.5 Killer Sudoku

3.5.1 What is a Killer Sudoku?

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

Figure 3.1: A Standard Sudoku puzzle. Taken from Wikipedia [17].

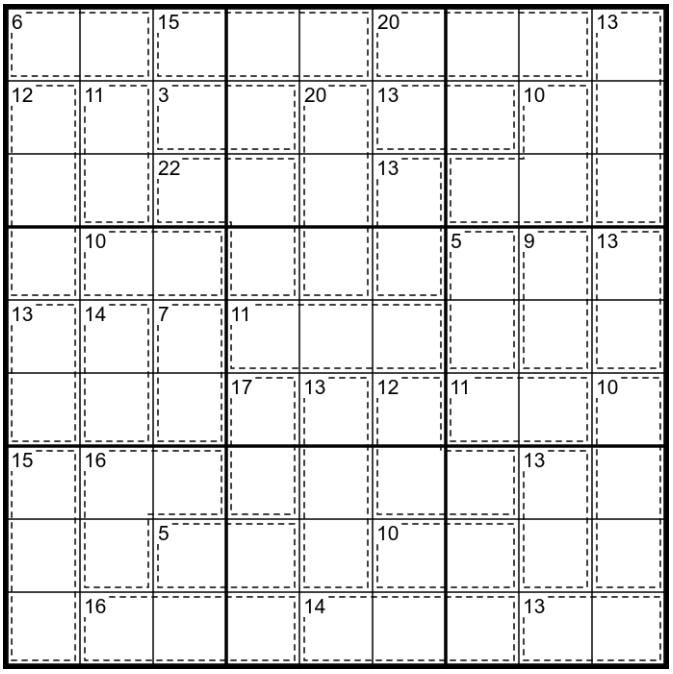


Figure 3.2: A Killer Sudoku puzzle. Taken from Daily Killer Sudoku, puzzle ID 349 [18].

Killer Sudoku puzzles are a variant of Sudoku puzzles that add additional rules, or constraints [19]. These additional rules also feature in Kakuro puzzles, which is another Japanese logic puzzle. We will refer to Sudoku as **Standard Sudoku** to avoid confusion.

Figure 3.1 shows a Standard Sudoku puzzle. To describe Standard Sudoku puzzles we firstly need some terminology:

- **Grid** - The whole puzzle
- **Cell** - A single square that contains one number in the grid
  - **Blank cell** - A cell that does not currently contain a number
  - **Clue cell** - A cell whose number is defined initially
- **Row** - A horizontal line of 9 cells
- **Column** - A vertical line of 9 cells
- **Box** - A 3x3 subgrid of cells, indicated by bold lines
- **House** - The collective term for Rows, Columns and Boxes

Each cell can be entered with any of the numbers 1-9. The grid contains 9 rows, 9 columns and 9 boxes. These make up 27 houses. To complete the puzzle, every blank cell must be entered such that all the houses contain all numbers 1-9. As can be seen in Figure 3.1, Standard Sudoku's feature clue cells. A *proper* sudoku puzzle will have only one solution [20].

Figure 3.2 shows a Killer Sudoku puzzle. Killer Sudoku puzzles have the same features as Standard Sudoku puzzles, but additionally have *cages*. A cage is a group of cells indicated by a dashed line. We will term the cells in a cage *cage cells*. Each cage contains a number, called the cage *clue*. Killer Sudokus do not feature any clue cells. The conditions for completing a Killer Sudoku are the same as Standard Sudoku with the additional rule that the values entered into the cells in each cage must sum to the cage clue and must be unique.

3.5.2 Human solving techniques

When a person solves a Standard Sudoku or a Killer Sudoku they start with an incomplete puzzle that contains blank cells. By applying a set of techniques the solver aims to enter every blank cell such that the rules of the puzzle are satisfied. We will refer to the value entered into a cell as the cell's *solid* value. When solving the puzzle the solver may note down all the possible solid values for a cell. We will refer to these values as the cell's *pencil* values. All techniques work on the same premise which is to eliminate pencil values. Once a cell has only a single pencil value, the solver enters it into the cell.

As all the rules of Standard Sudoku apply to Killer Sudoku, all Standard Sudoku techniques can be applied to Killer Sudoku puzzles. It is just as important to understand the Standard Sudoku techniques as it is the Killer Sudoku techniques when attempting a puzzle.

3.5.2.1 Standard Sudoku techniques

There are a huge range of Sudoku techniques with differing difficulties. Some Standard Sudoku techniques can be applied by observing solid values only. These techniques are considered

'easy', as the solver doesn't need to deduce any other information from the puzzle other than what they see in front of them. One example technique is scanning:

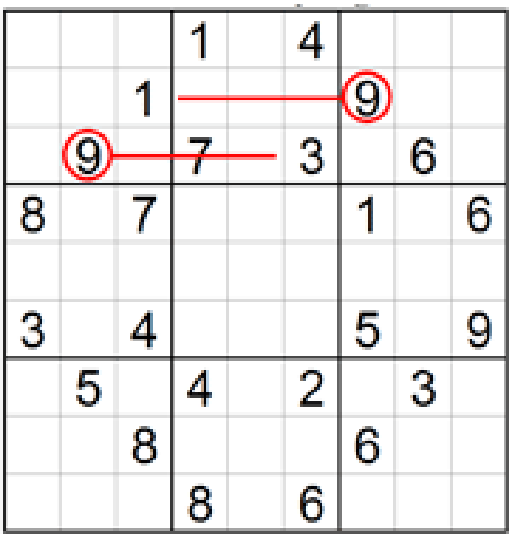


Figure 3.3: A house with one empty cell [21]. Image from `https://www.conceptispuzzles.com/`

Observe the top middle box. It can be deduced that the number 9 must go in the middle of the uppermost row, as it is the only cell in the box that can contain a 9 without violating the rules of the puzzle.

Other techniques require deducing information from the pencil values as well as the pen values. These techniques are considered more difficult to execute, as the solver first must make pencil markings before applying a technique. One of these techniques is 'naked pairs':

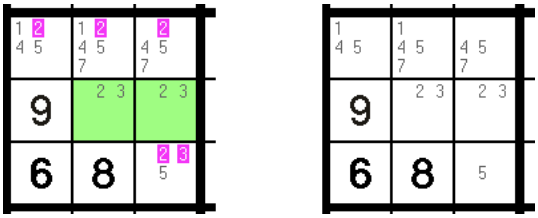


Figure 3.4: A box before and after naked pairs is executed. Image from `learn-sudoku.com` [22]

The box on the left hand side shows the state of a box before naked pairs has been applied. The cells highlighted green both have the same two pencil values, 2 and 3, and no others. From the Sudoku rules we know that the box must contain the number 2 once and the number 3 once. We are unable to work out which of the two cells contain 2 or 3, but we deduce that the other cells in the box cannot contain either 2 or 3. Therefore 2 and 3 can be removed from the pencil values of all other cells in the box. The resulting box is shown on on the right hand side. While naked pairs did not allow us to enter a solid value, it did allow us to eliminate pencil values.

More difficult techniques include X-Wing and Swordfish [23]. We will not go into the details of these. These techniques are considered more difficult because they require pencil values from more cells to be observed. The cells also may not be as obviously related to each other. For example, in X-Wing, 4 cells are observed, 2 in one row and 2 in another.

3.5.3 Killer Sudoku

3.5.3.1 The '45 rule'

Just as with Standard Sudoku, there are a range of Killer Sudoku techniques of differing difficulties. An easy Killer Sudoku technique involves exploiting the '45 rule'. The rules of the puzzle state that every house must contain the numbers 1 to 9. Therefore the numbers entered into every house must sum to 45 (as the numbers 1-9 sum to 45). This can be exploited to the solver's advantage. Application of the 45 rule is shown here:

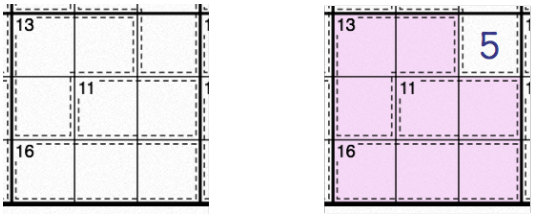


Figure 3.5: A box before and after the 45 rule is applied. Images taken from Daily Killer Sudoku's Strategies page [24].

The 8 cells in the 3 cages with clues 13, 11 and 16 must sum to 40 (13 + 11 + 16). These 3 cages are contained entirely within the box. The only cell that is not contained in any of these 3 cages is the cell in the top right hand corner. As we know the other 8 cells must sum to 40, and the 9 cells of the box must sum to 45, we know that the top right hand cell must contain the number 5 (45 - 40).

The 45 rule can also be used to generate *pseudo* cages. A pseudo cage is the same as a regular cage except that in some cases the cage cells can contain a number more than once. To derive pseudo cages we must introduce two new terms: *innie* cells and *outie* cells [25]. Innie and outie cells are relevant to a specific house. They are found by observing the cages in the house. If all a cage's cells are in the house then the cage does not contain any innie or outie cells. If any of a cage's cells are not contained within the house, then all its cells are either an innie or an outie. The innies are the cells that are in the house. The outies are the cells that are not in the house. This is best shown with an example.

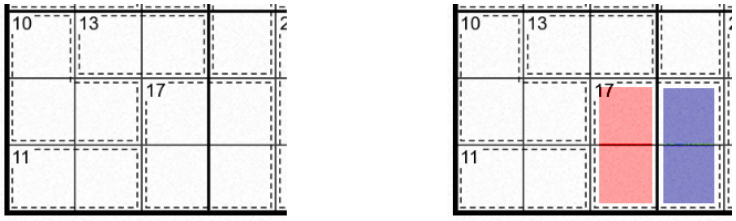


Figure 3.6: Deriving pseudo cages. Section of puzzle with ID 20044 taken from Daily Killer Sudoku [26].

The cage with clue 17 is the only cage with cells in the box that also has cells outside the box. The two cells in the cage that are in the box are the innie cells, shaded in red. The two cells in the cage that are outside the box are the outie cells, shaded in blue. Applying the 45 rule tells us that the innie cells must sum to 11 (45 - (10 + 13 + 11)). This creates the first pseudo cage. As the innie and outie cells are contained within the same cage with clue 17, and we know the

innies must sum to 11, the outie cells must sum to 6. This creates the second pseudo cage.

The '45 rule' can be extended and applied to groups of adjacent houses of the same type. We will term these house *blocks*. For example, the first two rows of the puzzle can be grouped together into a block. All the cells must now sum to 90, not 45. Groups of size 3 must sum to 135, and groups of 4 must sum to 180. Cages that contain cells inside and outside of the house block are now the cages containing innie and outie cells. The same logic as before can now be used to find pseudo cages.

3.5.3.2 Cage elimination

The number of cells in a cage and the cage clue can be used to eliminate pencil values. We will name this technique 'cage elimination'. These two pieces of information allow the solver to calculate the combination of values that the cage can permit such that all values are unique and sum to the cage clue. For example, lets say there is a cage with 2 cells and clue 6. There are only 2 combinations of 2 unique values that sum to 6: $(1, 5)$ and $(2, 4)$. We will call these the *cage combinations*. Therefore, each of the 2 cells can only be entered with one of the values 1, 2, 4 or 5. Any pencil value that is not one of these values can be eliminated. This logic can be applied to every cage, including pseudo cages. This is why generating pseudo cages is useful, as pseudo cages present more opportunities for eliminating pencil values. The difference when applying this logic to pseudo cages is that pseudo cages in some cases may permit non-unique values. Therefore the cage combinations also include combinations of non-unique values that sum to to the pseudo cage clue.

3.5.4 Puzzle difficulty

There is no official standard for the difficulty of Sudoku and Killer Sudoku puzzles. A website that publishes Sudoku puzzles for enthusiasts may rate a puzzle 'Intermediate', but if the same puzzle were published in a newspaper if may be rated 'Hard'. However, determining whether one puzzle is more difficult than another can be achieved by observing the techniques required to solve both puzzles. A puzzle that requires the use of harder techniques to solve it is a harder puzzle than a puzzle that only requires the use of easier techniques to solve it [27]. For example, a puzzle that can be solved by scanning only is considered easier than a puzzle that can only be solved by using naked pairs.

Even if a puzzle requires difficult techniques to solve it, the frequency that these difficult techniques will be used is small compared to the frequency that easy techniques will be used. Some puzzles may require the use of a particular technique only once to solve it. If successfully applied easier techniques also provide more useful information to a solver than harder techniques. For example, scanning may result in entering a solid value, whereas naked pairs can only eliminate pencil values. It is therefore a good strategy for a solver to execute the easier techniques before executing the harder techniques.

3.5.5 Daily Killer Sudoku

Daily Killer Sudoku, `https://www.dailykillersudoku.com/`, is a popular Killer Sudoku puzzle website. The site has over 20,000 puzzles. Puzzles are guaranteed to be proper [28]. Each puzzle is assigned a difficulty 1-10. Puzzles are taken from Daily Killer Sudoku for this study.

Chapter 4

Design and implementation

4.1 Design

The solver is designed as a constraint solver built specifically to solve Killer Sudoku puzzles. As with general purpose constraint solvers, the implementation of constraint propagation and search methods are core to the solver's design.

4.1.1 Expressing Killer Sudoku as a CSP

Firstly lets express Killer Sudoku a CSP using the formal definition (X, D, C) introduced in section 3.1.1. A Killer Sudoku contains 81 cells. Each cell corresponds to a variable. Each variable can be labelled 1-81, giving the set of variables $X = \{x_1, ...x_{81}\}$. The variables can instead be labelled more semantically with a row ID followed by a column ID. This makes it easier to identify which cell corresponds to which variable and vice versa. The letters $A - I$ are used to represent the rows and the numbers 1 - 9 are used to represent the columns. For example, x_{A6} corresponds to the cell in the 1st row, 6th column. Therefore, the set of variables, X, is equal to:

$$\{x_{A1}, x_{A2}, ..., x_{I8}, x_{I9}\}$$

A Killer Sudoku puzzle contains no clue cells, meaning every cell can be assigned to one of the numbers 1-9. Therefore the domain for every variable before solving begins is the same, and is equal to the set of numbers 1-9:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

After solving begins the domain of each variable will of course be filtered independently, so the domains will no longer be equal.

Before we express the rules of Killer Sudoku as formal constraints in the set C, lets express the rules of Standard Sudoku as constraints. A Standard Sudoku's rules state that all houses must contain the numbers 1-9. Equivalent, every cell in a house must be entered with a number 1-9, and all numbers in a house must be different. This can be expressed using a combination of `alldifferent` constraints [29]. The following constraint, $C_1 = (t_1, R_1)$, constrains all cells in row A to take different values:

$t_1 = \{x_{A1}, ..., x_{A9}\}$
$R_1 = \{(a, ..., i) : alldifferent(a, ..., i)\ \&\ (a, ..., i) \in D_{A1} \times \cdots \times D_{A9}\}$

A similar constraint is required for every other house in the puzzle, totalling 27 constraints. Killer Sudoku adds cages to Standard Sudoku, as described in section 3.5.1. To express Killer Sudoku as a CSP an additional constraint must be created for each cage. The constraint must constrain the cage cells to take unique values and sum to the cage clue. Lets create a constraint

for the cage containing the 3rd, 4th and 5th cells in the first row that has a cage clue of 15 in figure 3.2. As there exist 27 constraint already, lets number the constraint for this cage number 28. Our constraint, $C_{28} = (t_{28}, R_{28})$, is defined:

$t_{28} = \{x_{A3}, x_{A4}, x_{A5}\}$
$R_{28} = \{(a, b, c) : alldifferent(a, b, c)\ \&\ a + b + c = 15\ \&\ (a, b, c) \in D_{A3} \times D_{A4} \times D_{A5}\}$

A similar constraint is required for every cage. Killer Sudoku has now been formulated as a CSP. Solver design was influenced by this formulation. The solver implements three different types of solve. Rule solve executes human techniques on the puzzle. Search solve executes search techniques on the puzzle. MiniZinc solve models the puzzle and executes the MiniZinc solver.

4.1.2 Motivation for rule solve

Constraint propagation can be performed either by enforcing local consistencies or iterating reduction rules (i.e. executing propagators). The solver opts to implement propagators as its design is based off of general purpose constraint solvers, where propagators are the preferred method of constraint propagation.

When a person solves a Killer Sudoku they apply different techniques to eliminate pencil values. This equates to constraint propagation in CSP terms, and the techniques themselves equate to propagators. All Killer Sudoku puzzles no matter how hard their difficulty can be solved solely by human techniques without any guessing. All human techniques are derived from / exploit the rules of the puzzle. For example, scanning and naked pairs exploit the rule that each house must contain the numbers 1-9. We have shown that a Killer Sudoku's rules can be expressed formally with only two types of constraint. An `alldifferent` constraint for each house, and a constraint combining the `alldifferent` constraint with a sum constraint for each cage. A constraint solver therefore only requires two propagators to solve any Killer Sudoku, one for each type of constraint. A propagator is not required for each human technique. The filtering that would be performed by propagators for scanning and naked pairs would also be performed by a propagator for the `alldifferent` constraint, since both these techniques are derived from the rule that the `alldifferent` constraint implements.

Although a constraint solver does not require propagators for human techniques to solve a Killer Sudoku, this doesn't mean they shouldn't be implemented. Lets show this by examining a propagator for scanning. The propagator for `alldifferent` performs more filtering than the propagator for scanning, meaning it is a stronger propagator. As mentioned in section 3.2.2, stronger propagators usually have higher algorithmic complexity resulting in a higher cost of execution. An algorithm used commonly to implement the `alldifferent` propagator has complexity $O(m^2 \cdot d^2)$ [30]. The scanning propagator is quite simple, and should have a much lower running time than this. For a given house, the propagator must find all the solid values. The propagator must then remove each solid value from the pencil values of the incomplete cells in the house. The running time of executing scanning should therefore be lower than the

running time of executing the propagator for `alldifferent`. Rule solve is motivated by the fact that solving with propagators for human rules can outperform solving with general purpose propagators only. Of course if a puzzle requires more techniques than scanning to solve it, the scanning propagator would reach fixed point before the puzzle is solved.

4.1.2.1 Choosing human techniques to implement as propagators

The solver implements 5 propagators, each propagating a human technique:

- `elim_house()` - propagates scanning
- `elim_single_candidates()` - propagates single candidates
- `elim_naked_pairs()` - propagates naked pairs
- `find_pseudo_cages()` - propagates 45 rule
- `elim_cages()` - propagates cage elimination

All propagators except `find_pseudo_cages()` perform constraint propagation by filtering domains (eliminating pencil values). `find_pseudo_cages()` however adds new constraints (pseudo cages). `find_pseudo_cages()` is only called within `elim_cages()`, as the additional constraints are only beneficial for cage elimination. The other propagators are called independently.

4.1.2.2 Propagator scheduling

As described in section 3.2.2.1, one method of propagator scheduling is event directed scheduling, which involves implementing events and having propagators subscribe to them. The solver implements two types of event. One type is triggered by a change to a cell's pencil values. `elim_single_candidates()` and `elim_naked_pairs()` subscribe to events of this type. The second event type is triggered by a change to a cell's solid value. The previous two propagators as well as `elim_house()` subscribe to events of this type. The second event type was implemented because the propagator `elim_house()` only bases its eliminations on solid values, and does not consider pencil values at all. Therefore it is wasteful to subscribe `elim_house()` to an event triggered by a change to cell's pencil values.

The propagator `elim_cages()` does not subscribe to events and is instead queued when the queue is empty. When the solver was being developed initially an extremely naive method of propagator scheduling was used. All propagators were placed into a loop, which would exit if no eliminations occurred during a complete iteration of the loop. This is wasteful because some propagators will execute even their input variables are unchanged since they last executed, meaning it is impossible for them to make any eliminations. Each propagator implemented applying a human technique to the whole puzzle. All propagators except `elim_cages()` were then modified to only apply a rule a specified house. This was to allow implementation of scheduling. Due to its complex implementation `elim_cages()` was much harder to modify in this manner, so it was not modified.

4.1.3 Search solve design

Three search techniques are implemented for search solve: backtrack, backjump and forward checking. Generate and test is not implemented because it is too inefficient. Backtrack was chosen to be implemented because it is the simplest search technique to implement. Backjump was chosen to be implemented because it attempts to address issues with backtrack, so comparing the two algorithms is interesting. Forward checking was chosen to implemented as it is a hybrid method, and so is fundamentally different to backtrack and backjump.

4.1.4 MiniZinc solve design

MiniZinc solve was designed to encode a puzzle in the MiniZinc language whatever state it may be in (for example, unsolved or nearly solved) and solve it with the backend Gecode solver.

4.2 Implementation

4.2.1 Technology

The solver is implemented in Python. Development was extremely iterative and features were developed in small chunks. The development work for a feature could change the overall architecture drastically. Development of this style favours Python because of the speed at which functionality can be implemented and Python's high flexibility. Other languages such as C and C++ offer better control over memory than Python, which can result in higher efficiency when performing certain tasks. However, Python was opted for because its flexibility outweighed the advantages of other languages.

The code repository is on by GitLab. In early stages of development all development was made on the 'master' branch. However, feature branches were then used. Before each feature was developed an 'issue' was created. A branch was then created for the issue. Once the development work for the feature had been completed, the branch was merged back into master and the issue marked as complete. Making changes to the master branch directly was avoided.

The solver is run from the command line. The code was written in the PyCharm IDE [31]. Three additional programs are required to operate the solver, MiniZinc [32], Firefox [33] and geckodriver [34].

4.2.2 Code structure

In early prototypes simple data structures such as lists and tuples were used to represent elements of Killer Sudoku puzzles such as cells and pencil values. Human techniques were implemented as standalone functions that took these data structures as input. While this approach was satisfactory when implementing Standard Sudoku, when it came to implementing Killer Sudoku there were a lot of problems. It became difficult to remember which data structure represented each element and which indexes held required values. An OOP approach was adopted to solve these issues. Classes were created to implement different elements of puzzles.

The project structure follows guidelines given by Jean-Paul Calderone is his guide 'Filesystem structure of a Python project' [35].

4.2.3 Implementation of rule solve, search solve and MiniZinc solve

4.2.3.1 Rule solve implementation

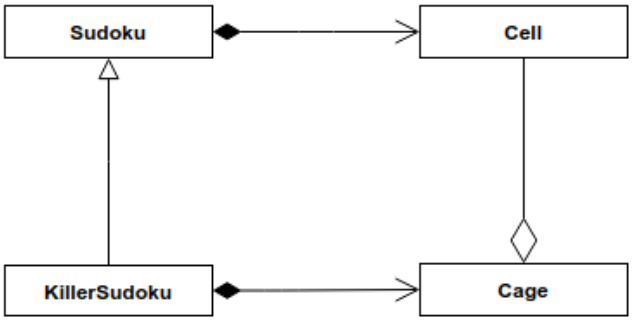


Figure 4.1: Class diagram

4 main classes are used for puzzle representation: Sudoku, KillerSudoku, Cage and Cell. The class diagram is shown in figure 4.1. Only relationships are shown, as there are too many fields and methods to display. Understanding these fields and methods is not necessary to understand rule solve at a high level. Standard Sudoku is implemented by the Sudoku class. Each Sudoku object has 81 Cell objects (one for each cell in a puzzle). The propagators `elim_houses()`, `elim_single_candidates()` and `elim_naked_pairs()` are methods of Sudoku. Each of these propagates a human technique on a given house. For example, calling `elim_naked_pairs("row", 0)` propagates the human technique 'naked pairs' on the 1st row (indexes start at 0).

Killer Sudoku is implemented by the KillerSudoku class. KillerSudoku extends Sudoku, so propagators of Sudoku can also by executed on KillerSudoku objects. Each KillerSudoku object has a number of Cage objects. Each Cage object has a reference to a number of Cell objects. Each Cage has a `clue` field for storing the cages clue. Propagators `find_pseudo_cages()` and `elim_cages()` are methods of KillerSudoku. `find_pseudo_cages()` has two parameters, `houses_block_count` and `max_cells_in_cage`. `houses_block_count` determines the size of the house block to use when finding pseudo cages, as described in section 3.5.3.1. `max_cells_in_cage` determines the maximum number of cells allowed in a pseudo cage. For example, if `max_cells_in_cage` was set to 2 the method would only return pseudo cages with 1 or 2 cells. Pseudo cages with more than 2 cells would not be created. The higher the values of these parameters, the more pseudo cages will be created and hence the more constraint propagation performed. However, the running time also increases. The parameters therefore allow the balance between constraint propagation and running time to be adjusted.

Rule solve can be instructed to end execution by specifying a value for timeout or minimum 'remaining domain values' (RDVs). If neither are specified rule solve will execute until the

propagators have reached a mutual fixed point. If a timeout is specified, after executing a propagator rule solve checks if the time elapsed has exceeded the timeout value. If it has, rule solve terminates. RDVs are the total number of remaining pencil values for uncompleted cells. They give a measure of the amount of filtering that has been performed. When a Killer Sudoku is created, there are 729 RDVs, as there are 81 uncompleted cells each with 9 pencil values. When the puzzle is solved there are 0 RDVs, as there are no uncompleted cells. A puzzle with between 0 and 729 RDVs is partially solved. If a minimum number of RDVs is specified, rule solve will check after executing each propagator whether the puzzle's RDVs has fallen below the minimum number specified. If they have, execution will stop. A puzzle's RDVs are inversely proportional to the amount of filtering, and hence constraint propagation, that has been performed. Stopping execution when RDVs falls below a certain number controls how much constraint propagation is performed on the puzzle.

Rule solve returns True if the puzzle has been solved. Rule solve returns False if rule solve could not solve the puzzle, meaning the propagators reached a mutual fixed point. Rule solve returns 'timed_out' if the given timeout value was reached before rule solve finished execution. Rule solve returns 'min_rdv_reached' if the number of RDVs falls below the number given.

`elim_cages()` implements the human technique cage elimination described in section 3.5.3.2. For a given cage, `elim_cages()` must calculate the cage combinations. Initially the combinations were found by inputting both the cage's clue and the number of cells in the cage into an algorithm. This algorithm, `find_cage_combinations_unique_numbers()`, can be found in 'Killer_Sudoku_Solver/killer_sudoku_solver/core/classes/loaders/CombinationsFileGenerator.py'. Due to the algorithms high complexity, its running time accounted for a significant proportion of the solver's total running time. To address this issue every possible cage combination was generated and put into a file 'combinations_unique.csv'. A class CombinationsLoader has a method to load this file and store the cage combinations in memory. Now whenever `elim_cages()` requires cage combinations to be generated it calls a method in CombinationsLoader which performs a simple lookup in memory and returns the cage combinations. This reduces solve time drastically.

4.2.3.2 Search solve implementation

The 3 search techniques, backtrack, backjump and forward checking, are implemented as standalone functions that do not belong to any class. Each search technique has its own file that contains two functions. The first function implements the search technique and the second function is a 'test' that is called by the first function to check the consistency of an assignment. The test function has a different implementation for each search technique because each technique requires different information from the test. Backtrack only needs to know whether an assignment is consistent or not. Backjump needs to know for an inconsistent assignment which variable caused the inconsistency. Backjump's test could also be used as backtrack's test, with backtrack ignoring the extra information. However, backjump's test function has a slower running time than backtrack's. For this reason the test is not reused, as backtrack's running

time would be increased unnecessarily. The test performed by forward checking is fundamentally different to both backtrack and backjump (as it involves enforcing arc consistency), so it must have its own test function.

Two variable orderings are implemented. The first ordering, named 'default' orders cells in the order they are defined in. This is left to right, top to bottom when looking at a puzzle. The second ordering, named 'fail-first', orders cells by the number of their pencil values, in increasing order.

Each search function takes a KillerSudoku object and a timeout value as input, and either outputs True if a solution was found, False if search exhausted and did not find a solution, or 'timed_out' if the timeout value was reached before search could finish execution.

4.2.3.3 MiniZinc solve implementation

MiniZinc models are implemented using the MiniZinc language. To find a solution to a model, the MiniZinc code is either written to a .mzn file and executed by the MiniZinc program or executed in the MiniZinc IDE. MiniZinc can be integrated directly into Python using the 'MiniZinc Python' library. Firstly a MiniZinc Model object is initialised and then the code implementing the model is added to it using the `add_string()` method. This code is exactly the same code that is used for modelling the problem in .mzn files and the MiniZinc IDE. Fortunately Standard Sudoku is modelled in the document 'A MiniZinc Tutorial' provided by MiniZinc [15, p .30]. The additional constraints for Killer Sudoku are added to this model. A constraint is required for each cage that specifies the cage cells and the cage clue. A method of KillerSudoku `generate_minizinc_constraints()` returns all the cage constraints in MiniZinc code. This code is added to the model. An 'instance' object is created from the Model object by specifying a constraint solver, which in this case is Gecode. A call to `solve()` on this object attempts to solve the model and returns a result.

4.2.4 Testing

A puzzle was tested for completion by a method of KillerSudoku `solved()`. The method returned True if the puzzle was solved, and False if it was not. This method was used to make sure that the propagators were correct. It was also used to make sure that when search solve or MiniZinc solve indicated a solution had been found, the solution was valid. Unit tests are developed for the propagators of Sudoku. These tests helped ensure the correctness of the propagators.

4.2.5 Obtaining and processing puzzle data

Daily Killer Sudoku does not offer a way of downloading Killer Sudoku puzzles in a computer readable format. A tool was built to pull puzzle data from the website and convert it into a .csv file. Each puzzle on Daily Killer Sudoku has its own page with URL 'https://www.dailykillersudoku.com/puzzle/<PUZZLE ID>'. Through inspection of the website code, it was discovered that a browser renders the puzzle using data in the

'window.DKS.puzzle' JavaScript object. The data is not stored in HTML. The tool uses Selenium to access this data. Selenium is a webdriver that is mainly used for automated testing. The tool uses Selenium to open the URL for a specified puzzle in Firefox in headless mode, meaning the browser window is not visible. A JavaScript function is then executed that pulls relevant data from the 'window.DKS.puzzle' object. This data includes the puzzle difficulty, and the co-ordinates and clues for every cage. The data is then converted into .csv format and output to a file, which we call the the 'puzzle file'. For example, if a puzzle has a cage with cells at co-ordinates (0,2) and (0,3) and a clue of 5, a line `"0|2,0|3",5` would be present in the puzzle file. The puzzle file is named 'dks_<PUZZLE ID>.csv'.

4.2.6 Workflow

A single execution of the solver is named a 'solving run'. A solving run firstly loads a Killer Sudoku puzzle, then executes different types of solve on the puzzle (rule solve, search solve and MiniZinc solve). The types of solve executed and the options used when executing each solve are determined by the 'solving configuration'. The solving configuration contains options such as the Killer Sudoku puzzle to be solved, whether to use rule solve and the timeout for a search strategy. The full configuration specification can be found in appendix B. Results are collected during the solving run, such as the run time of different solve types. All results collected can be found in appendix C.

Figure 4.2 shows the workflow of a solving run. Each time a type of solve is executed, it is run with the options specified in the configuration. For example, when the search solve is executed it will be run with the search technique and the search timeout specified in the configuration. A key point of note is that rule solve cannot be executed after any other type of solve. It is either the first type of solve executed by the solver or it is not executed at all. Another key point is that MiniZinc solve and search solve cannot follow one another. Only one of these solve types can be executed on each solving run.

A SolvingRun class implements a solving run. A SolvingRun object is initialised by specifying a solving configuration as a dictionary. The dictionary is saved to the objects 'config' field. During SolvingRun's execution results are collected in a field 'results', which is also a dictionary. SolvingRun has methods to call the 3 types of solve. The class contains a method `load_ks()` that takes as input the name of a puzzle file. The method creates a Cage for every line in the file, and then initialises a KillerSudoku object with these cages. The KillerSudoku object is then saved to the 'config' dictionary. The class has methods to call the three types of solve.

4.2.7 Operating the solver

The solver is run by executing 'python Killer_Sudoku_Solver/bin/ks_solve.py', with a command line argument. Figure 4.3 shows the screen displayed by running with argument `--help`. Option `-s` runs the solver in mode 1. The purpose of this mode is to run a one off solve for a particular puzzle and receive visual feedback of the solvers status and performance. Console output when running the solver in this mode is shown in figure 4.4. The solver prompts the user for the name of the puzzle to be solved and configuration options on the command

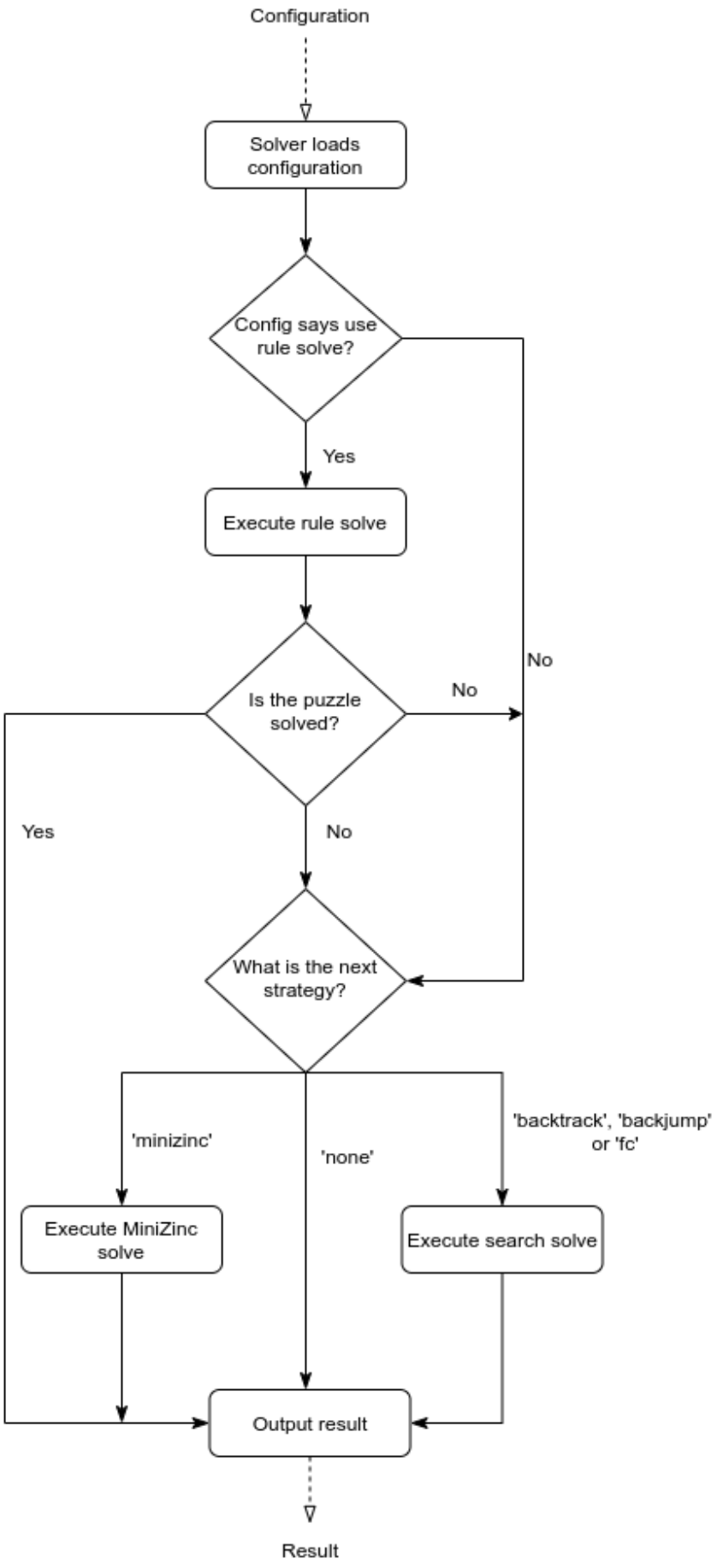


Figure 4.2: Workflow of the solver

```
Killer Sudoku Solver

Usage:
    python ks_solver.py -s
    python ks_solver.py -m <location of settings file>
    python ks_solver.py -d
    python ks_solver.py -l
    python ks_solver.py --help

Options:
    -s       Run solver specifying solving configuration in the console (mode 1)
    -m       Run solver specifying solving configurations in the settings file (mode 2)
    -d       Run tool to download puzzles from Daily Killer Sudoku
    -l       List the names of downloaded puzzles
    --help   Shows this screen
```

Figure 4.3: Solver help screen

```
Enter the name of the Killer Sudoku to be solved: dks_2744
Execute rule solver? (yes/no): yes
End condition for rule solve timeout, minimum remaining domain values or none? (t/rdv/none): t
Enter timeout in ms for rule solver: 200
Enter maximum blocks to use when finding pseudo cages: 4
Enter maximum cells of pseudo cages and cages processed by cage elimination: 4
Next strategy (backtrack/backjump/fc/minizinc): backjump
Enter timeout for search solve in ms or 0 for no timeout: 0

Performing rule solve...
Rule solver timed out, empty cells remaining:

0 0 0   0 0 0   0 0 0
0 0 0   0 0 7   0 0 0
0 7 1   0 0 9   0 0 0

1 9 7   0 4 0   0 0 0
0 0 0   0 0 0   0 0 1
0 0 0   0 0 1   4 0 0

0 0 5   0 0 0   0 0 0
0 0 0   0 0 0   0 0 0
0 0 0   0 0 4   0 0 0

Performing backjump search...
Search solve found a solution:

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

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

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

Timings:

Rule solve time: 206ms
Search time: 2737ms
Total solve time: 2943ms
```

Figure 4.4: Output when running the solver in its first mode

line. The name of downloaded puzzles can be shown by running the solver with option `-l`. Once these are given solving begins. If rule solve is executed during the solving run the state of the puzzle is shown after rule solve finishes. If another solve is executed, the state of the puzzle is shown again after that solve finishes. After the final solve finishes, the program outputs the final state of the puzzle, execution times for each solve type and the total solve time.

Running the solver with option `-m` runs the solver in its second mode. Multiple solving configurations are created and run. Results are output to a .csv file. Processing the data in the .csv files allows the solver's performance to be analysed when running different solving configurations. To use the solver in this mode a settings file must be supplied after the `-m` argument. A sample settings file exists in 'python Killer_Sudoku_Solver/bin/' called 'settings.py'. The command `python ks_solve.py -m settings.py` runs the solver in mode 2 with this settings file. The settings file is processed and multiple SolvingRun objects are created. The settings file contains all the options required for a solving run, but lists of values are given for each option rather than single values. A configuration is created for every combination of values. For example, if in the settings file `ks_name` was set to ['dks_14422', 'dks_19830'], `next_strategy` was set to ['backtrack', 'backjump'] and every other option was set to a list containing only a single value, 4 configurations would be created, with `ks_name` and `next_strategy` set to the following in each configuration:

- `ks_name`: 'dks_14422', `next_strategy`: 'backtrack'
- `ks_name`: 'dks_14422', `next_strategy`: 'backjump'
- `ks_name`: 'dks_19830', `next_strategy`: 'backtrack'
- `ks_name`: 'dks_19830', `next_strategy`: 'backjump'

The created configurations are filtered of redundant configurations. For example, for the following settings 4 configurations would be created:

`next_strategy`: ["backtrack", "minizinc"]
`search_timeout`: [3000, 5000]

One of these configurations would set `next_strategy` to "minizinc" with `search_timeout` set to 3000, and another would set `next_strategy` to "minizinc" with `search_timeout` set to 5000. However, `search_timeout` is an irrelevant option as the next strategy is not a search strategy. These two configurations instruct the solver to execute in exactly the same way, so one of them is redundant and is removed. Once all redundant configurations have been removed, a SolvingRun object is created for every configuration, added to a queue and then executed in turn.

After every solving run has finished execution the solver outputs the results to a .csv file. Each line in the csv file is the combined configuration and result for a solving run. By default, the data produced by every solving run will be output to one file. However, an option in the settings file allows a separate file to be produced for each puzzle. Each file is timestamped with

the date and time. The results output are the same as those output by the solver when run in its command line mode. The output files are placed in 'Killer_Sudoku_Solver/killer_sudoku_solver/data/results'.

The solver comes pre-loaded with some puzzles. If the user wishes to download more they can run the puzzle downloading tool by running the solver with option `-d`. The user is prompted for puzzle IDS separated by commas. The tool will download the corresponding puzzles from Daily Killer Sudoku. Daily Killer Sudoku also publishes 'greater-than' Killer Sudoku puzzles. These puzzles are not compatible with the solver, and will not be downloaded. The names of the new puzzle files that the tool has generated are shown in the console.

Chapter 5

Analysis

The solver offers a huge number of solving possibilities. Analysis was limited to a few key aspects:

- Performance of rule solve against the Gecode solver
- Performance of search methods
- Performance of MiniZinc solve when running `find_pseudo_cages()` propagator beforehand
- Performance of forward checking with different variable orderings

Experiments were designed that ran range of solving configurations and compared the results. The solving configurations were setup by creating a settings file for each experiment and running the solver in mode 2. Analysis could then be performed by processing the data in the output .csv file. For each experiment a script was created to process the data. The settings file, the results file and processing script for each experiment can be found in the repository. For each experiment the location of the files within the repository is given. All puzzles used in the experiments come pre-downloaded with the solver.

Note that all the results files except the results file for the 'Performance of rule solve against the Gecode solver' experiment are missing the result `gecode_solve_time`. All experiments except this one were run on a build that did not calculate this result. Other than not calculating this result, the builds were the same.

Experiments were all run on a desktop PC with the following specification:

- OS: Linux Mint 18.1 Cinnamon 64-bit
- CPU: Intel Core i5-3570k @3.4Ghz
- GPU: Palit GeForce Nvidia GTX 660 Ti 2GB
- RAM: 8GB DDR3 1333MHz

5.0.1 Performance of rule solve against the Gecode solver

Settings file: 'settings_and_results/exp_1/exp_1_settings.py'
Results file: 'settings_and_results/exp_1/exp_1_results.csv'
Processing script: 'settings_and_results/exp_1/exp_1_process.py'

As MiniZinc solve solves using the Gecode solver, solving configurations using MiniZinc solve allow for analysis on the performance of the Gecode solver. All puzzles in this experiment have

a maximum difficulty of 7, as rule solve alone is unable to solve the vast majority of puzzles with difficulty higher than 7. The settings file sets up 17 solving configurations for each of the 100 puzzles specified. 16 of these configurations run rule solve only, and differ by the values set for options `rule_solve_max_blocks` and `rule_solve_max_cells`. Lets label these configurations as type 'A' configurations. The remaining configuration runs MiniZinc solve only. Lets label this configuration as type 'B'. For a given puzzle, if a type A configuration runs in less time than the type B configuration, then rule solve outperforms MiniZinc solve for that puzzle. For type A configurations, the run time was taken as the value of `rule_solve_time`. For type B configurations the run time was taken as the value of `gecode_solve_time`. `minizinc_solve_time` was not used as it includes the time taken to model the current puzzle. In this case we want to compare the propagation schemes of rule solve and the Gecode solver, so the time taken to model the problem should not be accounted for in the run time. Rule solve was able to solve 83 of the 100 puzzles. Of these 83 puzzles, a type A configuration outperformed the type B configuration on 64 puzzles. Therefore, rule solve outperformed the Gecode solver on 64 of the 100 puzzles.

Section 4.1.2 describes the motivation for developing rule solve, explaining that propagators developed for human techniques can outperform stronger general purpose propagators. This result supports this reasoning. Rule solve was able to outperform the Gecode solver, which implements stronger general purpose propagators, for the majority of the tested puzzles. Of course, rule solve was only able to outperform the Gecode solver on puzzles that could be completed with rule solve only. If the goal was to minimise run time but guarantee finding a solution, the Gecode solver is clearly the best choice of solver, as it will always find a solution no matter how high the puzzle difficulty. However, if rule solve was further developed to implement propagators for more advanced human techniques, the choice between the two types of solve may not be so simple.

5.0.2 Performance of search methods

Settings file: 'settings_and_results/exp_2/exp_2c_settings.py'
Results file: 'settings_and_results/exp_2/exp_2c_results.csv'
Processing script: 'settings_and_results/exp_2/exp_2c_process.py'

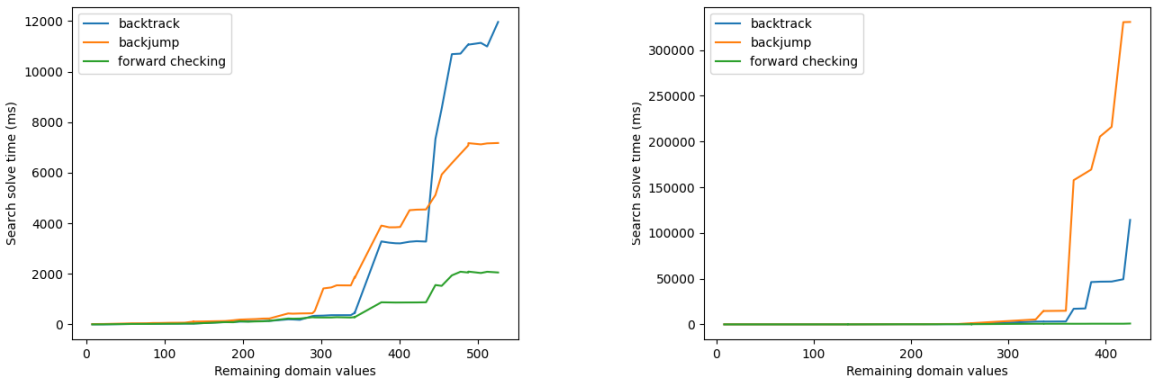
The performance of the 3 implemented search methods (backtrack, backjump and forward checking) was analysed by comparing the run times of each method when solving different puzzles in a range of states. A puzzle's state is determined by its RDVs. Therefore a range of RDVs must be used to obtain a puzzle in a range of states. This is achieved by setting the option `rule_solve_end_condition` to 'rdv' and `rule_solve_min_rdvs` to a range of values. Each search method is then run on the resulting puzzle states. For a given search method, it is expected that the higher the number of RDVs, the higher the run time of search. This is because RDVs are inversely proportional to the amount of constraint propagation performed, and we know from section 3.3.3 that the more constraint propagation performed, the larger the increase in efficiency of search.

Each search algorithm was tested on 100 puzzles in 3 different states. The 3 different states were obtained by setting the minimum number of RDVs to 150, 250 and 350. A timeout was set at 10 seconds. When solving the puzzles with 150 RDVs backtrack was the best algorithm, outperforming backjump and forward checking on 55 puzzles. 10 puzzles could not be solved by any algorithm within the time limit. When solving the puzzles with 250 RDVs forward checking was the best algorithm, outperforming backtrack and backjump on 57 puzzles. 10 puzzles could not be solved by any algorithm within the time limit. When solving the puzzles with 350 RDVs forward checking was again the best algorithm, outperforming backtrack and backjump on 60 puzzles. 13 puzzles could not be solved by any algorithm within the time limit. These results show that for puzzles with 150 RDVs backtrack is the best algorithm, and for puzzles with 250 and 350 RDVs forward checking is the best algorithm. It is very likely that forward checking is the best algorithm for puzzles with 250 and more RDVs, but these results are not enough to categorically prove it.

This can be shown further by observing the running time of each algorithm on specific puzzles in a higher number of states. This can be achieved by setting a larger range of minimum RDVs at smaller intervals. Plots in figure 5.1 were generated by this method. These are the settings and results files for each plot:

Daily Killer Sudoku puzzle with ID 15893:
Settings file: 'settings_and_results/exp_2/exp_2a_settings.py'
Results file: 'settings_and_results/exp_2/exp_2a_results.csv'
Processing script: 'settings_and_results/exp_2/exp_2a_process.py'

Daily Killer Sudoku puzzle with ID 8229:
Settings file: 'settings_and_results/exp_2/exp_2b_settings.py'
Results file: 'settings_and_results/exp_2/exp_2b_results.csv'
Processing script: 'settings_and_results/exp_2/exp_2b_process.py'



(a) Daily Killer Sudoku with puzzle ID 15893

(b) Daily Killer Sudoku with puzzle ID 8229

Figure 5.1: Performance of search algorithms on two different puzzles

In plot **a** the running time of each algorithm is similar up to around 280 RDVs. As the RDVs increases past 280 the running time of each algorithm begins to diverge. At 500 RDVs the run time for backtrack, backjump and forward checking is around 11500ms, 7000ms and 2000ms respectively. In plot **b** the running time of each algorithm begins to diverge at around 300 RDVs. The relative increase in the run time of forward checking is so small compared to the increase of backtrack and backjump's run time that it doesn't appear to increase at all.

As forward checking performs better than backtrack and backjump for higher numbers of RDVs, it shows that for high numbers of RDVs the added cost of enforcing arc consistency is lower than the cost saved by identifying dead-ends sooner. For lower numbers of RDVs the opposite is true.

5.0.3 Performance of MiniZinc solve when running `find_pseudo_cages()` propagator beforehand

Settings file: 'settings_and_results/exp_3/exp_3_settings.py'
Results file: 'settings_and_results/exp_3/exp_3_results.csv'
Processing script: 'settings_and_results/exp_3/exp_3_process.py'

As described in section 4.2.3.3 the method `generate_minizinc_constraints` models the loaded puzzle in the MiniZinc language. Without calling `find_pseudo_cages()`, only the constraints imposed by the original cages of the puzzle are modelled. After calling `find_pseudo_cages()` the constraint imposed by each pseudo cage can also be added to the model. Each pseudo cage can be modelled by an additional sum constraint. An additional `alldifferent` constraint can also be added to the model for a pseudo cage if the cage's cells share a house. MiniZinc solve can benefit from these additional constraints.

An experiment was setup to analyse the extent of this benefit. 100 random puzzles were downloaded from Daily Killer Sudoku. Two types of solving configurations were run on each puzzle: One that ran `find_pseudo_cages()` before MiniZinc solve, and one that ran MiniZinc solve only. Lets label them type 'C' and type 'D' respectively. `find_pseudo_cages()` has two parameters, `houses_block_count` and `max_cells_in_cage`. Each can be set with the configuration options `minizinc_max_blocks` and `minizinc_max_cells`. For this experiment the run time was taken as the value of `total_solve_time`. Therefore the execution of `find_pseudo_cages()` and modelling the current puzzle are accounted for in the run time. For 82 of the 100 puzzles the lowest run time was achieved by a solving configuration of type C. For 69 of these configurations, the run time was decreased by between 0.1% and 50% compared to the type D configuration. The largest decrease in run time was from 22884ms to 162ms, a decrease of 99.3%. This was when solving puzzle with ID 17301.

The 82 type C configurations that achieved the lowest run time on a puzzle did not have the same values set for `minizinc_max_blocks` and `minizinc_max_cells`. The optimal parameters for `minizinc_max_blocks` and `minizinc_max_cells` changed from puzzle to puzzle. Therefore, without knowing the optimal values of these parameters, it is better to execute a type B solve.

The fact that MiniZinc solve alone can be beaten by running `find_pseudo_cages()` before MiniZinc solve is interesting nonetheless. Further analysis could be performed to determine the optimal values for these parameters.

These results highlight the importance of understanding the structure of a problem. By studying Killer Sudoku puzzles, humans have developed techniques to exploit the puzzles structure. When a person attempts a Killer Sudoku puzzle, they know that applying the '45 rule' to create pseudo cages is a useful technique. Whilst modelling a Killer Sudoku puzzle with a collection of `alldifferent` and sum constraints is perfectly sufficient for a constraint solver to find a solution, these constraints alone do not allow a constraint solver to exploit the puzzles structure. As the '45 rule' technique is problem specific, a constraint solver is very unlikely to have a dedicated propagator for it. It is not within the scope of this project to understand the propagation scheme of Gecode. However, it is safe to say by looking at these results that Gecode's propagation scheme can be improved with the addition of the `find_pseudo_cages()` propagator.

5.0.4 Performance of forward checking with different variable orderings

Settings file: 'settings_and_results/exp_4/exp_4_settings.py'
Results file: 'settings_and_results/exp_4/exp_4_results.csv'
Processing script: 'settings_and_results/exp_4/exp_4_process.py'

The performance of different variable orderings were measured in a similar way to the performance of different search algorithms. Puzzles of different states were obtained by setting minimum RDVs to 150, 250 and 350. A forward checking search was then run twice on each puzzle state, once with default variable ordering and once with fail-first variable ordering. A timeout was set for 10 seconds. When solving the puzzles with 150 RDVs the default cell ordering was the best ordering, outperforming fail-first ordering on 62 puzzles. 10 puzzles could not be solved by any algorithm within the time limit. When solving the puzzles with 250 RDVs default cell ordering was again the best ordering, outperforming fail-first ordering on 59 puzzles. 10 puzzles could not be solved by any algorithm within the time limit. When solving the puzzles with 250 RDVs default cell ordering was again the best ordering, outperforming fail-first ordering on 57 puzzles. 11 puzzles could not be solved by any algorithm within the time limit.

These results show that the default variable ordering is better than the fail-first ordering for puzzles with 150, 250 and 350 RDVs. It is likely that the default variable ordering is better for puzzles with RDVs between 150-350, but these results are not enough to categorically prove it. This is a bit surprising at first, as we know from section 3.3.2 that the heuristic implemented for fail-first performs well for randomly generated problems. Section 3.3.2 mentions another heuristic for realising the fail-first principle is to choose the next variable as the one which has the most constraints with assigned variables. As the default ordering orders cells row by row and all cells in a row are constrained against each other, the default ordering for cells in the puzzle is actually quite close to the ordering that would be created by this other heuristic. There is also an overhead when ordering with fail-first caused by having to reorder the cells

after every assignment. These two facts makes this result less surprising.

Chapter 6

Conclusion

6.1 Summary

The aim of this project was to express Killer Sudoku as a CSP and evaluate the performance of different constraint satisfaction methods for solving it. This has been achieved to some extent. We have introduced the area of constraint satisfaction, allowing Killer Sudoku to be modelled as a CSP. A solver motivated by the introduced theory was built. The solver implements a range of constraint satisfaction methods, and allows comparisons of their performance to be made.

The introduction of constraint satisfaction begins with formally defining a CSP, which is key to understanding constraint satisfaction methods. Constraint propagation was introduced, giving an overview of the most common techniques. These include enforcing local consistencies and rules iteration. Global constraints were also introduced, with an explanation of their importance not only in modelling but solving too. 3 common search techniques, backtrack, backjump and forward checking were introduced. A brief background of constraint solvers was given.

Understanding the theory behind constraint satisfaction methods was key when designing the solver. We have firstly expressed Killer Sudoku as a CSP. We went on to design the three different solve types, rule solve, search solve and MiniZinc solve. We set out to design the rule solve as a problem specific constraint solver, involving the implementation of propagators to achieve constraint propagation. We showed that implementing propagators for human techniques is important in increasing efficiency. We explained how event directed scheduling could be implemented by creating two event types, one for changes to a cell's pencil values and one for changes to a cell's solid value.

The solver was written in Python due to the language's flexibility. The solver opted for an object-oriented design so Standard Sudoku and Killer Sudoku could be modelled easily. 5 propagators are successfully implemented. 3 as methods of Standard Sudoku and 2 as methods of Killer Sudoku. The 3 search methods backtrack, backjump are implemented as standalone functions. MiniZinc solve is implemented by encoding the constraints of a specific Killer Sudoku in the MiniZinc language, and solved using the the MiniZinc Python library. A tool was successfully implemented to download puzzles from Daily Killer Sudoku in a computer readable format. This improved testing and analysis significantly, as hard-coding a puzzle is a very time consuming task.

We have explained that a solving configuration is used to determine the solver's behaviour. The solver can be operated in two different modes. The first mode builds a single solving configuration by taking input from the console. The state of the puzzle and solving times are

printed to the console after solving has finished. The second mode builds multiple solving configurations from a given settings file, executes each configuration and outputs the results to .csv file. Processing the results in the .csv file allows analysis of the performance of the solver.

Analysis of the solver's performance highlighted a few key points. Backtrack was found to be the best search algorithm for puzzles with 150 RDVs, and forward checking was found to be the best algorithm for puzzles with 250 and 350 RDVs. For puzzles that rule solve could solve, rule solve outperformed MiniZinc solve. This enforced the importance in developing propagators for human techniques. Executing the `find_pseudo_cages` propagator before running MiniZinc solve was found to outperform MiniZinc when run on its own in some cases. This showed that the `find_pseudo_cages` propagator improved the propagation scheme of the Gecode solver, and showed the importance of understanding a problem's structure.

6.2 Future work

Whilst this project has explored solving Killer Sudoku as a CSP, a lot of areas touched on could be further developed.

There are some key improvements that can be made to the solver. The first is to improve the rule solver by implementing more human techniques. Analysis showed the importance of developing propagators implementing human techniques. This could be proven further by developing more of them. Ideally, propagators could be implemented for every known technique. This is a huge task however. The more difficult human techniques require a lot skill to develop and test. Although the development of more propagators is likely to improve the propagation scheme of the solver, the time complexity of propagators increases as the difficulty of the human technique implemented increases. Therefore execution of these additional propagators may have a negative impact on running time.

The solver could implement more sophisticated propagator scheduling. For example, propagators could be prioritised once they are queued, so that propagators with the highest priority are executed ahead of lower priority propagators. The current event based scheduling could also be improved. For example, a new event could be implemented that is triggered when the count of a cell's pencil values is equal to 2. Naked pairs would then subscribe to events of this type only, as naked pairs can only perform filtering when there are at least 2 cell's with exactly 2 pencil values.

Further search methods could be developed, such as Maintais Arc Consistency (MAC), which is a hybrid algorithm [5, p .89]. Forward checking was proven to be the best search algorithm of the 3 implemented for 250 and 350 RDVs. It would be interesting to compare forward checking against another hybrid algorithm.

6.3 Legal, Social, Ethical and Professional issues

6.3.1 Legal issues

Two very minor legal issues arose during the project. The first was the inclusion of MiniZinc code from the MiniZinc documentation that models Standard Sudoku. This was appropriately referenced in the report and in the code. The second issue was obtaining puzzle data. All puzzle data was downloaded from Daily Killer Sudoku. Daily Killer Sudoku is free to access, so appropriate referencing addresses this issue also.

6.3.2 Social issues

Expressing Killer Sudoku as a CSP and solving it with different constraint methods has no social implications.

6.3.3 Ethical issues

No ethical issues arose during the project. However, it is possible that the results found could improve unethical applications that use constraint satisfaction methods. As the results discovered by this project are not groundbreaking this is very unlikely.

6.3.4 Professional issues

Only one person developed this project, so no professional issues concerning client and shareholder interaction arose. The work of others is correctly referenced. Testing was performed to make sure results obtained were accurate.

6.4 Self Assessment

This project has taught me the importance of performing background research before development. Understanding the human methods for solving Killer Sudoku and the theory of constraints satisfaction methods was key to implementing the solver.

The solver is a complex and sophisticated tool. Development drew heavily on my software development skills learned throughout university and my placement year. The solver is able to output meaningful data that allows the comparison of constraint satisfaction methods. I think the fact that the propagation scheme developed for rule solve was able to outperform the propagation scheme used by the Gecode solver is impressive. In early stages of development I did not believe this would be possible, as the run time for rule solve was enormous compared to the run time of rule solve in the current build of the solver. Unfortunately due to time constraints rule solve could not be developed further.

This project has also taught me the importance of following a methodology closely. By following the agile methodology I was able to build the solver incrementally. This allowed early integration of the different solve types, allowing early comparisons of solve types to be made. I have learnt however not to neglect unit tests. The method/propagator `find_pseudo_cages()`

required a significant amount of development time. The method calls several other methods. Developing unit tests for each of these called methods would have reduced the amount of time required to develop `find_pseudo_cages()`.

References

[1] Roman Barták, Miguel Salido, and Francesca Rossi. New trends in constraint satisfaction, planning, and scheduling: A survey. *Knowledge Eng. Review*, 25:249–279, 09 2010.

[2] Wikipedia. *Sudoku*. [Online]. 2019. [Accessed 17 December 2019]. Available at: `https://en.wikipedia.org/wiki/Sudoku`.

[3] Jyoti1, Tarun Dalal. Constraint satisfaction problem: A case study. *International Journal of Computer Science and Mobile Computing*, 4(5):33–38, 2015.

[4] I. Miguel and Q. Shen. Solution techniques for constraint satisfaction problems: Foundations. *Artificial Intelligence Review*, 15(4):243–267, 2001-06.

[5] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., USA, 2006.

[6] Nikos Samaras and Kostas Stergiou. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *CoRR*, abs/1109.5714, 09 2011.

[7] Guido Tack. *Constraint Propagation Models, Techniques, Implementation*. 2009.

[8] Mathematical Programming Glossary. *Global constraint*. [Online]. No date stated. [Accessed 5 May 2020]. Available at: `https://glossary.informs.org/ver2/mpgwiki/index.php?title=Global_constraint`.

[9] Willem-Jan van Hoeve. *The alldifferent Constraint: A Survey*. *CoRR*, cs.PL/0105015, 05 2001.

[10] Jean-Charles Régin. *Global Constraints: A Survey*, pages 63–134. 11 2010.

[11] Barbara M. Smith. *A Tutorial on Constraint Programming*. 1995.

[12] Roman Barták. CONSTRAINT PROPAGATION AND BACKTRACKING-BASED SEARCH. [Online]. 2005. [Accessed 8 May 2020] Available at: `https://ktiml.mff.cuni.cz/~bartak/downloads/CPschool05notes.pdf`.

[13] Dimitris Vrakas and I. Vlahavas. *Artificial intelligence for advanced problem solving techniques*. 01 2008.

[14] Roman Barták. Value and Variable Ordering. [Online]. 1998. [Accessed 5 May 2020]. Available at: `https://ktiml.mff.cuni.cz/~bartak/constraints/ordering.html`.

[15] Kim Marriott and Peter J. Stuckey. *A MiniZinc Tutorial*. [Online]. No date stated. [Accessed 6 May 2020]. Available at: `https://www.minizinc.org/tutorial/minizinc-tute.pdf`.

[16] MiniZinc. *MiniZinc Python*. [Online]. 2019. [Accessed 5 May 2020]. Available at: `https://minizinc-python.readthedocs.io/`.

[17] Wikipedia. *A typical Sudoku puzzle ...* [Online]. 2020. [Accessed 8 May 2020]. Available at: `https://en.wikipedia.org/wiki/Sudoku`.

[18] Daily Killer Sudoku. *Puzzle 349.* [Online]. 2009. [Accessed 5 May 2020]. Available at: `https://www.dailykillersudoku.com/puzzle/349`.

[19] Wikipedia. *Killer sudoku.* [Online]. 2019. [Accessed 10 December 2019]. Available at: `https://en.wikipedia.org/wiki/Killer_sudoku`.

[20] Wikipedia. *Glossary of Sudoku.* [Online]. 2020. [Accessed 5 May 2020]. Available at: `https://en.wikipedia.org/wiki/Glossary_of_Sudoku`.

[21] Conceptis Ltd. *Scanning in one direction A.* [Online]. No date stated. [Accessed 5 May 2020]. Available at: `https://www.conceptispuzzles.com/picture/27/1186.gif`.

[22] learn-sudoku.com. *naked_ pair2.* [Online]. 2008. [Accessed 5 May 2020]. Available at: `https://www.learn-sudoku.com/images/naked_pair2.gif`.

[23] learn-sudoku.com. *Advanced Techniques.* [Online] 2008. [Accessed 5 May 2020]. Available at: `https://www.learn-sudoku.com/advanced-techniques.html`.

[24] Daily Killer Sudoku. *Strategies.* [Online]. 2020. [Accessed 5 May 2020]. Available at: `https://www.dailykillersudoku.com/strategies`.

[25] https://www.sudokuwiki.org/. *Innies and Outies.* [Online]. 2008. [Accessed 7 May 2020]. Available at: `https://www.sudokuwiki.org/Innies_And_Outies`.

[26] Daily Killer Sudoku. *Puzzle 20044.* [Online]. 2020. [Accessed 5 May 2020]. Available at: `https://www.dailykillersudoku.com/puzzle/20044`.

[27] Conceptis Ltd. *Conceptis Sudoku difficulty levels explained.* [Online]. 2006. [Accessed 5 May 2020] Available at: `https://www.conceptispuzzles.com/index.aspx?uri=info/article/2`.

[28] Daily Killer Sudoku. *Rules.* [Online]. No date stated. [Accessed 5 May 2020]. Available at: `https://www.dailykillersudoku.com/`.

[29] Helmut Simonis. *Sudoku as a Constraint Problem.* 2005.

[30] Basileios Anastasatos. *Propagation Algorithms for the Alldifferent Constraint.* 05 2020.

[31] Jet Brains. *Pycharm.* [Online]. 2020. [Accessed 8 May 2020.] Available at `https://www.jetbrains.com/pycharm/`.

[32] MiniZinc. *MiniZinc.* `https://www.minizinc.org/`.

[33] Mozilla. *Firefox.* `https://www.mozilla.org/en-GB/firefox/`.

[34] Mozila. *geckodriver.* `https://github.com/mozilla/geckodriver`.

[35] Jean-Paul Calderone. *Filesystem structure of a Python project.* [Online]. 2007. [Accessed 6 May 2020] `http://as.ynchrono.us/2007/12/filesystem-structure-of-python-project_21.html`.

Appendices

Appendix A

External resources

The only external resource used for this project is the model of Standard Sudoku in the MiniZinc language. This can be found in the document 'A MiniZinc Tutorial' [15, p .30]. This code is used in the method `minizinc_solve()` in file `Killer_Sudoku_Solver/killer_sudoku_solver/core/SolvingRun.py`.

Appendix B

Solving configuration specification

The below options are required when specifying a solving configuration. For each option, the possible options are given, then a description of how the option adjusts the solving run.

`output_option`: 'one_file' or 'file_per_puzzle'. If set to 'one_file' all data from all solving runs will be output to one file. If set to 'file_per_puzzle' the data for each puzzle will be output to separate files.

`ks_name`: The name of the killer sudoku instance to be solved, for example 'dks_14422'. Do **not** include '.csv'.

`use_rule_solve`: True or False. If set to True rule solve is used. If set to False it is not.

`rule_solve_end_condition`: 'rdv', 't', 'none'. Determines when rule solve should stop execution. If set to 'rdv', rule solve will stop when the remaining domain values count falls below a certain number. This number is specified by setting option `rule_solve_min_rdv`. If set to 't' rule solve will stop after a certain number of milliseconds have elapsed. The number of milliseconds is set in option `rule_solve_timeout`.

`rule_solve_min_rdv`. Integer 0-729. Only applicable if `rule_solve_end_condition` is set to 'rdv'. Once the remaining domain values count falls below this number, rule solve will terminate.

`rule_solve_timeout`: Integer. Only applicable if `rule_solve_end_condition` is set to 't'. The number set for this option is the maximum time in milliseconds that rule solve can execute for before terminating and returning status 'timed_out'.

`rule_solve_max_blocks`: Integer 1-4. Sets 'max_blocks' parameter for `elim_cages()` propagator.

`rule_solve_max_cells`: Integer. Sets 'max_cells_in_cage' parameter for `elim_cages()` propagator.

`next_strategy`: 'backtrack', 'backjump', 'fc', 'minizinc'. The solve strategy to use after rule solving. 'fc' stands for forward checking.

`search_timeout`: Integer. The maximum time in milliseconds a search strategy (backtrack, backjump or forward checking) can execute for before terminating and returning status 'timed_out'.

`cell_order_option`: 'd' or 'f'. Only applicable if `next_strategy` is set to 'fc'. Sets the variable ordering. Variable ordering is described in section 3.2.2. Options are 'd' for default and 'f' for fail-first.

`minizinc_gen_pseudos`: True or False. If set to True, the `find_pseudo_cages()` propagator will run before MiniZinc solve. If set to False the propagator will not be run. However, this option is only applicable if `use_rule_solve` is False, as if rule solve is executed pseudo cages will already have been found.

`minizinc_max_blocks`: Integer 1-4. Only applicable if `minizinc_gen_pseudos` set to True. `find_pseudo_cages()` run before MiniZinc solve will find pseudo cages with house blocks of size 1 to the value set. For example, if set to 3 then pseudo cages will be found using house blocks of size of 1, 2 and 3.

`minizinc_max_cells`: Integer. Only applicable if `minizinc_gen_pseudos` set to True. Sets 'max_cells_in_cage' parameter of `find_pseudo_cages()` that is run before MiniZinc solve.

Appendix C

Results output

The below values are saved to SolvingRun's dictionary 'result'. The values are output to the console if running the solver in mode 1. These values are combined with the configuration options and output to a .csv file if running the solver in mode 2. For each result, the possible values are given, then a description of the meaning of each value.

`rule_solve_status`: True, False, 'timed_out', 'min_rdv_reached'. If True, rule solve succeeded in solving the puzzle. If False, rule solve's propagators reached a mutual fixed point but the puzzle was not solved. If 'timed_out', rule solve exceeded the timeout specified in the solving configuration.

`rule_solve_rdvs`: Integer. The remaining domain values after rule solver execution ended.

`rule_solve_time`: Integer. The time elapsed in milliseconds by rule solve.

`search_solve_status`: True, False, 'timed_out'. If True, search solve solved the puzzle. If False, the search algorithm finished execution without finding a solution. If 'timed_out', search solve exceeded the timeout specified in the solving configuration.

`search_solve_time`: Integer. The time elapsed in milliseconds by search solve.

`minizinc_solve_status`: True or False. If True, MiniZinc solve solved the puzzle. If False, MiniZinc failed to solve the puzzle.

`minizinc_solve_time`: Integer. The time elapsed in milliseconds by MiniZinc solve. This includes modelling of the current puzzle, creation of the MiniZinc instance and the call to `solve()` on the MiniZinc instance.

`gecode_solve_time`: Integer. The time elapsed in milliseconds of `solve()` called on a MiniZinc instance.

`total_solve_time`: Integer. The time elapsed by all solve types.

Appendix D

Software Repository

The repository can be found at the following URL:

`https://gitlab.com/sc16hd/sc16hd_project`