University of Glasgow | School of Computing Science

Animating a Sudoku Solver

Gabriel Ionut Stratan

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 25, 2016

Abstract

This report introduces and explains general concepts and algorithms from Constraint Programming that were used throughout the project. The project consists of developing an animated visualization of the *all-different* algorithm introduced by Jean-Charles Régin [11] in 1994. The Sudoku puzzle was chosen to illustrate the algorithm as it represents a simple enough problem that uses a great number of constraints of difference for the rows, columns and sub-squares. A full implementation of the *all-different* algorithm is provided in this report, which can be easily adapted to solve problems such as scheduling, timetabling and supply chain management. Finally, this report concludes with the implementation of the Java program that will be used by lecturers and students to visualize how this powerful algorithm works.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ____________________ Signature: ____________________

Contents

Chapter 1

Introduction

The Sudoku puzzle is a well known combinatorial problem where the goal is to fill in a partially completed 9×9 grid with digits from 1 to 9 in such a way that a single digit cannot appear twice in the same row, column or 3×3 sub-square. The puzzle is a type of a Latin square that was shown to be NP-complete by Charles Colbourn [2] and will be used as an example to introduce the concepts of the all-different algorithm used in Constraint Programming to solve many hard combinatorial problems. Students new to Constraint Programming will be shown a new way of approaching and solving real-world problems by thinking in terms of constraints that impose restrictions on what values variables can take.

1.1 Aims

The aim of this project is to visualize how Constraint Programming techniques can be used to solve Sudoku puzzles. A Java program is developed to act as a teaching aid to be used by the lecturer to show students how the all-different algorithm works towards a solution. Sudoku puzzles represent an excellent choice to introduce the concept of the all-different constraint in an engaging experience with the students. Students will be shown how to model the Sudoku puzzle as a Constraint Satisfaction Problem and get to the solution without resorting to guessing or brute forcing. The constraint programming technique that will be used to solve the Sudoku puzzle is the same algorithm used worldwide to solve planning, scheduling, timetabling and supply chain management problems. The step by step visual representation of how the algorithm progresses towards a solution will help students better understand the new algorithm.

1.2 Background

The project will provide a visualization for the all-different algorithm introduced by Jean-Charles Régin [11] in 1994. The step by step visualization will show how a Sudoku puzzle is solved using the algorithm. Although the original 9×9 Sudoku puzzle can be solved in a fraction of a second using a variety of algorithms, the power and effectiveness of the all-different algorithm used in this report can be seen when trying to solve bigger Sudoku grids, where the solution is found in a matter or seconds, rather than hours or days using a less efficient approach [13]. The all-different algorithm is considered an important milestone in the Constraint Programming field. The algorithm's efficiency comes from pruning the search space of a problem of combinations that break the constraints of the problem. Before the introduction of this algorithm, the standard approach in the field used binary constraints that had a poor performance.

1.3 Motivation

The motivation of this project is to provide a tool that makes it easier for a lecturer to introduce the all-different algorithm to students by providing a step by step view of how the algorithm progresses. Students will find it easier to remember and understand the algorithm through the use of the visualizations. The tool based on the Sudoku puzzle is a great way to visualize an important technique used to solve Constraint Satisfaction Problems and will provide an engaging learning experience to the students.

1.4 Report Content

The rest of this report will provide information on the algorithms used and their implementation in the project.

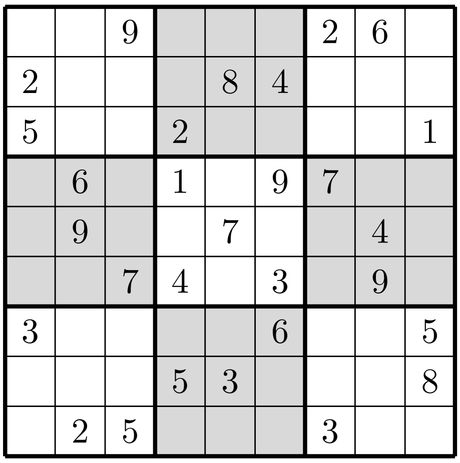
- **Chapter 2** covers details about the Sudoku puzzle and Constraint Programming.
- **Chapter 3** introduces a powerful constraint of difference.
- **Chapter 4** explains the two algorithms needed to solve a constraint of difference.
- **Chapter 5** covers a demonstration of the project solving a Sudoku puzzle.
- **Chapter 6** explains the implementation of the project.
- **Chapter 7** details the overall achievements of the project.

Chapter 2

Background

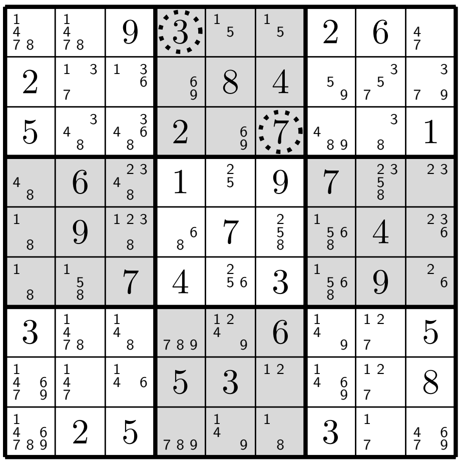
2.1 Sudoku

Sudoku, meaning *single number* in Japanese, is a logic-based, combinatorial puzzle that became mainstream in 1986. Today, many magazines and newspapers from all over the world include a Sudoku puzzle. The aim of the game is to place digits from 1 to 9 into a 9 × 9 grid, such that the same digit may not appear twice in the same row, column or 3 × 3 sub-square of the puzzle. A Sudoku puzzle starts as a partially completed grid that has to be filled by the player. Enough clues are provided such that the puzzle has a unique solution. An example of a relatively hard Sudoku puzzle is shown in 2.1. Variations of the puzzle include instances featuring $n^2 \times n^2$ sized boards with $n \times n$ sub-squares. Since the Sudoku puzzle is known to be NP-complete [2], this means that a general Sudoku requires significantly more computational resources as n increases.



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

Fig. 2.1: A Sudoku puzzle



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

Fig. 2.2: The state of the puzzle after the propagation

One effective way of solving Sudoku puzzles is by using techniques and algorithms used in Constraint Programming. The puzzle can be modelled as a Constraint Satisfaction Problem [12] that has 81 variables and 27 all-different constraints, one for each of the 9 rows, column and 3 × 3 sub-squares of the puzzle. Each constraint in the Sudoku puzzle represents a relationship between 9 variables that make up a row, column or 3 × 3 sub-square. Variables that represent clues given in the initial puzzle have a domain that contains a single value, while the unknown variables have the set of digits from 1 to 9 as their domain of possible values. Multiple techniques are then used to get to a solution by removing infeasible values from the domain of the variables until each one of them is left with only a single value in their domains, representing the correct assignment in the solution.

Propagation schemes are used to achieve consistency of the constraints. This means that for each variable that has been solved (has a single digit in their domain of possible values), one can remove that value from the domain of the variables that take part in the same constraints. In the case of the Sudoku puzzle, each variable is part of 3 all-different constraints, one for the row it takes part in, one for the column, and one for the 3×3 sub-square. Propagation is done starting from the variables representing clues in the initial puzzle, and can trigger iterative calls once other variables have their domain reduced to a single value. The iterative calls stop when no more values can be removed from any of the variables' domains. Easy Sudoku puzzles that have many clues [6] can be solved by using this technique alone.

2.2 shows the state of the initial Sudoku puzzle after the propagation is performed. One can notice that the clues provided in the initial puzzle were enough to find out the correct assignment for two variables (circled in the figure). The rest of the variables had their domains reduced to keep the constraints consistent. Notice the first column that contains two variables that have $\{1, 8\}$ as their domain. This is the place where a human may resort to guessing, and where an intelligent algorithm should be used so one can find a solution to the puzzle. The all-different algorithm notices that values 1 and 8 should be assigned to the two variables at a later stage in the solution. This is enough information to decide that neither 1 nor 8 can be taken by the rest of the variables in the column. This will result in the variable that has $\{4, 8\}$ as its domain being assigned the value 4 and will inform the other variables in the same column that they cannot take the value 4. Since this is known, the variable that had $\{1, 4, 7, 8\}$ as its domain will be force to take the value 7 as the rest of the values were previously assigned to other variables. By applying the same all-different algorithm on rows, columns and 3×3 sub-squares, one will eventually find the unique solution without having to resort to any guessing. Since the Sudoku puzzle is an NP-complete problem [2], there is no way to know beforehand which row, column or sub-square to solve next. The advantage of the algorithm is that the search space for a solution is pruned by removing values that will never be part of the solution, making the search much faster.

2.2 Constraint Programming

Constraints can be found everywhere in our daily life, and represent mathematical abstractions of the dependencies in the physical world. More formally, a constraint is a logical relationship among a number of variables, each with its own domain. The constraint imposes restrictions on the values the variables can take. It is good to note that the values of the variables are not always numeric, a heterogeneous constraint would require a word (string) to have a specific length (numeric). Constraints are declarative as they only specify a relationship that must hold between variables and do not provide an algorithm to enforce the relationship.

Constraint Programming is the area of Computing Science that solves problems by specifying a list of constraints and then finding solutions that satisfy all the constraints. A Constraint Satisfaction Problem (CSP) can be defined as a triple $\mathcal{P} = \{X, D, C\}$, where X represents a set of n variables $X = \{x_1, x_2, ..., x_n\}$, D represents the n domains associated to the n variables $D = \{D_1, D_2, ..., D_n\}$ such that $x_i \in D_i$, and finally C represents a set of t constraints $C = \{C_1, C_2, ..., C_t\}$. A constraint C_i restricts the values one or more variables can simultaneously take.

A solution for a CSP $\mathcal{P}$ is a set of values $S = \{s_1, s_2, ..., s_n\}$, such that each variable gets assigned a value from its domain and all the constraints are satisfied at once. The set of all the solutions to the problem is noted as $sol(\mathcal{P})$ for a CSP that has multiple solutions. One may also be interested in finding an optimal solution to the problem, one that for example minimizes the path of a traveling salesman. When $sol(\mathcal{P})$ is equal to the empty set, it means that the original CSP is not satisfiable. This simple representation can be used to model complex real world problems such as planning, scheduling, timetabling, supply chain management and more.

There are three strategies for solving CSPs: inference, search and learning. If all the variables in the problem have finite domains, this means the search space for a solution is finite and represented by all the possible combinations of the values of the variables. Although one can, in theory, enumerate all the possible combinations, we use inference and search to reduce the search space. Inference techniques remove large subspaces of the search tree through local constraint propagation on the basis that no solutions may be found using a particular value from the domain of a variable. Search techniques, such as backtracking, are used to systematically explore the search space, usually with the use of some heuristics, and reduce the search space whenever a single failure is found. Both strategies are frequently used together and work because a solution to a CSP must have all its constraints satisfied, therefore a local inconsistency on a subset of the variables will not result in a solution.

The simplest backtracking search algorithms look for solutions to a CSP by traversing the search tree in a depth-first search manner. At each node of the search tree, a variable is assigned a value from its domain, and the node gets extended with branches for the remaining values in the variable's domain that need to be considered later, as they could be part of a solution. For example, consider a simple CSP problem with three variable x, y and z. Suppose that the search algorithm already assigned a value to the variable x out of its domain, and it reaches a node representing the variable y with the domain $D_y = \{1, 2, 3\}$. The backtracking search algorithm will generate three branches, each one of them representing the variable y getting instantiated to a value from its domain. For every value assigned to the variable y, all the constraints tied to this variable and the previously visited variables are checked to see if they still hold. When the constraints are not satisfied by a selected value for a variable, the algorithm proceeds in assigning and testing the rest of the values in the domain of y. Once all the values from the domain of the variable y have been tried, the algorithm will backtrack to the previous variable, x in this case, and assign it the next value from its domain. Note that every time backtracking occurs, subtrees of the search tree will no longer be generated and visited since they will not contain any solutions to the CSP. In the problem mentioned above, the subtree for the values of z is no longer generated and tested as the partial solution made out of some specific values for x and y already violated the constraints of the problem. A complete solution is found once all the variables are assigned values that are allowed by the constraints. If the user is interested in only one solution to the problem, the search will terminate, otherwise the algorithm will continue to explore the search tree for other solutions. In the case the problem has no solutions, the algorithm will terminate after visiting all the possibilities in the search tree.

The efficiency of the backtracking search algorithm can be improved by performing constraint propagation to maintain consistency between the values of the variables and their associated constraints. Every time a variable's domain is reduced, constraint propagation is performed on the constraints the variable was part of. Propagation will check all the variables that were in a constraining relationship with the original variable and will remove values from their domain that are inconsistent with the problem requirements. By reducing the domains of the variables, the search tree is pruned and the efficiency of the search increases. The domain of a variable may be reduced to the empty set after propagation, which means that there is no value left for that variable that will satisfy the constraints and therefore backtracking should be initiated. On other occasions, the domain of a variable will be reduced to a single value, which means that the value for the variable is now known and removing the need to search through values that would not lead to a solution. The backtracking search algorithm will resume when the propagation results in no changes to the domains of the variables. Information from constraint propagation is usually incorporated in the search algorithm in form of variable and value ordering heuristics that improve the efficiency of the search even more.

Variable and value ordering heuristics are ways to improve the backtracking search algorithm by prioritizing the instantiation of some variables and by choosing some values from their domain first. Ordering heuristics are essential to effectively solve hard combinatorial problems. The most popular heuristic is the fail-first approach, something Haralick and Elliot [7] described as "To succeed, try first where you are most likely to fail". The principle is to prioritize the instantiation of variables with the smallest domains of values. This is done such that the branches in the search tree that have no solutions get discovered earlier and pruned to bring a noticeable increase in efficiency later in the search. Consider the worst-case scenario when performing a backtracking search on a problem that has no solutions, the search tree will be much smaller as earlier failure is encouraged, making it faster to prove that a problem has no solutions. In the case of problems that start off with equally sized domains, the fail-first principle can still be applied by prioritizing the instantiation of the variables that take part in many constraints. Once a variable was selected for instantiation, some values from its domain could lead to a solution faster than others. The order values are chosen for assignment to the variables is only relevant when trying to find a single solution to the problem. The reason behind this is that trying to find all the solutions to a problem or proving that there are no solutions would have to traverse all the search space anyway. A heuristic to prioritize values from a domain is to assess the likelihood each value has in getting closer to a solution, a choice that will have minimum impact on the domain of other variables after propagation, therefore increasing the chances of finding a solution faster.

Some solutions to a problem are better than others. Consider the NP-hard Traveling Salesman Problem (TSP), where the shortest possible path between some cities should be found. The general approach to the optimization problem is to introduce a constraint that minimizes a variable representing an objective, the length of the path in the case of the TSP. Finding an optimal solution is often hard, and proving optimality of a solution might be impossible for complex combinatorial problems.

Constraint Programming can be regarded as a form of declarative programming, where a user specifies a problem in terms of its variables, their domains and the constraints that should be satisfied. The computer then solves the given problem using a range of techniques that are general approaches to problem solving which can be applied to all the problems that can be modeled as constraint satisfaction problems.

The original exercise of finding an algorithm to solve a specific problem is now turned into an exercise of finding the best way to model the problem as a CSP, which can later be solved by the computer. The model can make the difference between efficient solving of a problem or a problem that cannot be solved due to its combinatorial complexity. There are usually many ways in which one can model the same problem, but the efficiency of the solution could be different. Consider a problem in which a number of g guests should be offered a seat at one of the t tables, with the constraint that couples should sit together. One could decide to represent each of the g guests as a variable associated with an initial domain representing a listing of the t tables, from which a choice for the table should be made based on the constraints. Another way to model the same problem is to have a number of t variables representing the tables, each with an initial domain representing a listing of all the guests. The search for a solution would reduce the domains of the variables in both models until there is only a single table in the domains of the variables (for the first model where guests are variables), or in the case of the second model, until all the variables representing tables have the cardinality of their domains reduced to their sitting capacity. Although both models are logically equivalent, one may perform worse than the other because of the way solvers work.

When modelling a problem, it is important to avoid symmetries in the form of equivalent solutions. Symmetries represent wasted computational resources during the backtracking search that is used to explore symmetrical assignments. One should avoid such symmetries by adding new constraints that will filter out the symmetrical solutions. Such constraints are problem specific. If it is necessary to find all the solutions to a problem, including the equivalent ones, one can easily perform permutations of the variables outside of the search space once the main solutions are found, thus avoiding exploring similar branches in the search tree. In the example problem mentioned above, there are symmetries in the solutions, as the same group of guests will be seated at different tables throughout the solutions. As we know tables are the same in real life and can be easily swapped without violating the constraints, such computations should not be made to improve efficiency of the search.

Real world problems are often large and over-constrained, which may raise difficulties in modelling them. Most of the time there are conflicting objectives such as trying to maximize the efficiency of a production facility, while minimizing the operational costs. Trade-offs between the constraints should be made to come up with a solution to the problem. Problem decomposition should also used to break a large real world scenario into smaller problems that are easier to solve. Finally, one should consider which model is easier to upgrade when more constraints should be added.

Chapter 3

The All-Different Constraint

3.1 Review of Jean-Charles Régin's paper

This section will introduce the reader to Jean-Charles Régin's paper [11] published in 1994, which describes a filtering algorithm for constraints of difference used in solving CSPs. As mentioned before, because the search for all the solutions to a CSP is NP-complete, there is a justified need to prune the search space both before and during the search for the solutions.

A constraint of difference is defined as a relationship on a subset of the variables of a problem for which the values that are assigned to them should be all different. Many real world problems require such constraints. The paper starts by recognizing the need for a powerful n-ary constraint of difference, as binary constraints of difference cannot efficiently prune the search space of values for variables that will never be part of a solution, and for most instances it does not delete any value. Before the publication of the paper, the common practice to represent the n-ary constraint of difference was to model it as many binary constraints, one for each pair of variables in the n-ary constraint.

Consider a CSP problem with tree variables x, y and z with their domains $D_x = \{1, 2\}$, $D_y = \{1, 2\}$ and $D_z = \{1, 2, 3\}$ with a constraint that solutions should have different values assigned to the variables. The first way one could model such a problem is by introducing 3 binary constraints of difference for all the pairs of the variables: $\{x, y\}$, $\{x, z\}$ and $\{y, z\}$. The second way, is to use a 3-ary constraint of difference between the variables $\{x, y, z\}$. The difference between the two approaches is that a propagation of the constraints for the binary representation will not reduce the domains of the variables. On the other hand, the model using a 3-ary constraint will successfully remove the value 1 and 2 from variable z, as they will already be used by x and y.

Jean-Charles Régin [11] goes on to describe a generalized constraint of difference to take advantage of the improved pruning of the search space. The aim is to achieve arc-consistency across all the variables in the constraint. A variable x is said to be arc-consistent with the other variables participating in the same n-ary constraint if for all the values in D_x, there exists admissible values in the domains of the other variables such that the constraint holds. For the purpose of this exercise, the aim is to achieve diff-arc-consistency, which is concerned with achieving arc-consistency across the constraints of difference.

A constraint of difference C can be represented by its value graph. The value graph of C is a bipartite graph where the variables in the constraint are grouped in the left partition of the graph, while the right partition represents a union of all the values from the domains of the variables. Edges between the two partitions of the graph connect the variables on the left-hand side with values from their domains.

Matches in the value graph of C are a subset of the edges in the graph, where no two edges share a common vertex. A *maximum match* is when all the vertices in the left partition have an edge to a vertex from the right partition, or more specifically, all the variables in the left partition are assigned a value from the right one. After finding maximum matchings for all the value graphs associated to the constraints of difference in a given CSP, the values assigned to the variables do not break the diff-arc-consistency for the problem. Since each variable is assigned a value from its domain represented by an edge in the graph, and no two variables are assigned the same value, the diff-arc-consistency for the problem still holds. Knowing this property, an algorithm is built to filter the domains of the variables by removing values that would never be part of a maximum matching, therefore breaking the constraint and not leading to any solutions to the problem. It is good to note that by using the notation mentioned earlier, all the constraints of difference can be represented by their value graphs that have a space complexity in $O(pd)$, where p is the arity of the constraint and d is the maximal cardinality of the domains for the variables in the constraint. Computing maximum matchings in a bipartite graph $G = (X, Y, E)$ with m edges, can be done using Hopcroft and Karp's algorithm [8] or the Ford-Fulkerson algorithm [4].

To begin reducing the domains of the variables, more definitions regarding matches are introduced. First, a vertex is $matched$ if there is an edge connecting it to a vertex in another partition of the graph. Vertices that are not matched are said to be *free*. *Alternating paths* represent paths made of edges that are alternatively matches and free and have a length corresponding to the number of edges in the path. *Vital edges* are edges that are part of all the maximum matchings in the value graph of a constraint.

Berge's property [1] is used to find the edges that are not in any maximum matchings and can therefore be deleted. According to the property, such edges are not vital and when starting from an arbitrary maximum matching, as no even alternating paths can be found to include that edge. The algorithm starts by computing a maximum matching for the value graph of a constraint of difference. After this, an algorithm such as the one proposed by Tarjan [14] is used to find the strongly connected components in the graph. This means that edges in such a component will belong to even alternating paths, as the graph is bipartite and does not have any odd cycles. The step of finding strongly connected components is necessary as this means that every edge connecting vertices from two different strongly connected components can be deleted as it will never be part of an even alternating path. The deletion of an edge in the value graph of the constraint represents the reduction of a variable's domain. The complexity of this algorithm proposed by Jean-Charles Régin [11] is of $O(p^2d^2)$ for one constraint of difference.

Reducing the domain of a variable by deleting edges in the value graph of a constraint can trigger modifications for other constraints involving the same variable. If a variable is part of other constraints of difference, the author shows a more efficient propagation approach than simply repeating the same algorithm. The filtering algorithm for the second constraint builds on the information from the previous step by trying to remove the same edges in the value graph of the new constraint as long as they are not vital and a new maximum matching can be found.

The paper goes on to solve the zebra problem, a famous logic puzzle. The author models the problem in different ways to prove that his approach can prune the search space of the problem more when compared to the previous methods used in the industry. One representation of the problem uses binary constraints for each pair of variables involved in the constraint of difference and another representation uses the n-ary constraint of difference introduced by the author.

The results of his experiment show that the search space of the problem gets pruned more using the approach proposed in the paper, making the search for a solution to the problem more efficient. The same method was successfully used in solving problems such as subgraph isomorphism.

3.2 Usage example

To demonstrate the usage and effects of the all-different constraint, a simple example that uses the all-different algorithm is provided below. The CSP problem will be modeled in Java and will make use of Choco3 [10], a powerful free and open-source Java library used to solve Constraint Satisfaction Problems. The implementation of the all-different constraint in the Choco3 library is based on Jean-Charles Régin's paper [11] reviewed above.

The problem has three variables x, y, z with the following initial domains: $x = \{1, 2\}$, $y = \{1, 2\}$ and $z = \{2, 3\}$. The constraint that should be satisfied by all the solutions of the problem is that x, y and z should all be different. The Listing 3.1 shows how the problem is modeled and implemented using the Choco3 library.

```
 1  Solver solver = new Solver(”AllDifferentExample”);
 2
 3  IntVar x = VF.enumerated(”x”, new int[]{1, 2}, solver);
 4  IntVar y = VF.enumerated(”y”, new int[]{1, 2}, solver);
 5  IntVar z = VF.enumerated(”z”, new int[]{2, 3}, solver);
 6
 7  solver.post(ICF.alldifferent(new IntVar[]{x, y, z}));
 8
 9  try{
10      solver.propagate();
11  } catch (ContradictionException e){
12      // Handle the exception
13  }
14
15  System.out.println(x + ”\n” + y + ”\n” + z);
16
17  if(solver.findSolution()){
18      do{
19          int x_val = x.getValue();
20          int y_val = y.getValue();
21          int z_val = z.getValue();
22
23          System.out.println(x_val + ”; ” + y_val + ”; ” + z_val);
24      } while(solver.nextSolution());
25  }
```

Listing 3.1: Usage of the all-different constraint in Choco3 [10]

Line 1 defines a Choco3 solver object, that will store the variables, their domains and constraints associated to them. Lines 3 to 5 define the three integer variables in the problem and their associated initial domains. The reference to the *solver* object makes sure the variables are added to the model. Line 7 introduces the all-different constraint between the three variables. The constraint is then posted to the model and the CSP is now fully modeled.

Before trying to find a solution, a call to the *propagate* method is performed on the model on line 10. The propagation algorithm will make all the constraints consistent by looping thorugh all the constraints in the model (only one in this case) with the aim of reducing the domains of the variables by removing values that will never part of a solution. When a value is removed from the domain of a variable, this might trigger an iterative propagation call on all its dependant variables to make the constraints consistent. The iterative propagation calls end when the problem is in a consistent state. It is good to note that propagation will preserve all the original solutions to the problem, while greatly reducing the search space.

The *ContradictionException* on line 11 is an exception that is raised by Choco3 whenever a variable is left with an empty domain or is instantiated to a value that is out of its initial domain. This usually happens in overconstrained problems, where there are no values in the domains of the variables that satisfy all the constraints at once. In such cases, there would be no valid solutions to the problem. This is not the case with the example showcased here, as there are three variables with three different values in total.

Once the propagation is finished, a call is made on line 15 to output the updated domains of the variables. The updated domains after the call to the *propagation* method are as follows: $x = \{1, 2\}$, $y = \{1, 2\}$ and $z = \{3\}$. The variable z lost the value 2 from its domain, while to the domains of x and y stayed the same. There are two reasons behind removing the value 2 from the domain of variable z. Firstly, there would no solutions with z taking the value of 2 as this would leave either x or y with an empty domain after one of them takes the value of 1. Secondly, two variables (x and y) have the same domain of $\{1, 2\}$, therefore a solution that is consistent with the all-different constraint will have x taking the value of 1 and y taking the value of 2 (or vice-versa), making it impossible for z to take the value of 2. Since we know that the values 1 and 2 will be attributed to x and y, the value 2 is removed from the domain of z to reflect this change and to reduce the search space.

Lines 17 to 25 will iterate over the two solutions of the problem and print the values for the variables. The solutions to the problem are $S_1 = \{1, 2, 3\}$ and $S_2 = \{2, 1, 3\}$ and they both contain values for the three variables that satisfy the all-different constraint.

Chapter 4

Algorithms for All-Different

4.1 Ford Fulkerson

This section introduces an algorithm used to compute a maximum matching in a graph. A *flow* represents a directed graph with special vertices called the *source* and the *sink*. Source vertices have outgoing edges to inner vertices, while sink vertices have incoming edges from inner vertices of the graph. The problem requires sending flows, for example water through a pipe, from the source to the sink, while conserving the equilibrium.

Flows need to get from the *source* to the *sink* by following the directed edges that often have different maximum capacity constraints, such as the diameter of a water pipe. Given a flow f in graph G, a residual directed graph G_f is generated. Forward edges in the graph have not reached their maximum capacity, while backward edges reached their upper bound.

The Ford-Fulkerson algorithm [4] uses the residual graph to achieve a maximum flow in the graph. The algorithm looks for augmenting paths from source S to sink T in a breadth first-search manner. Once a path p is found, the weight that will make the bottleneck edge in p fill its maximum capacity is sent. This is made by adding (subtracting) the weight for each forward (backward) edge in p. Forward edges that reach their maximum capacity turn into backward edges. This results in flow f' that has a greater value than before. Berge's theorem [1] guarantees that once all the augmenting paths have been successively visited in the algorithm, the end result is a maximum flow of the graph.

The maximum matching problem is a special case of the maximum flow problem. For the purpose of this project, there will be only one source S that has outgoing edges to the vertices in the left partition L of the bipartite graph and one sink T with incoming edges from the right partition R of the graph. Vertices in L are connected through edges to vertices in R. To find a maximum matching for a Graph G, the source and sink vertices are added to the graph and all edges are set to have a maximum capacity of 1. We then run the Ford-Fulkerson algorithm [4] on the graph, by finding all the augmenting paths from free vertices in L to free vertices in R.

For the purpose of this project, if no maximum matching is found in the value graph of the constraint, then one can backtrack immediatly as this means that the constraint cannot be satisfied. Having a maximum matching for our graph, the next step in the all-different algorithm is to turn the previously undirected graph, into a directed one. The matches resulted from the Ford Fulkerson algorithm [4] are assigned a direction from the 1 to 9 values to the corresponding cells of the Sudoku row. The edges that remain unused in the matching are given a direction from left to right (i.e. from the Sudoku cells of the row to the 1-9 values).

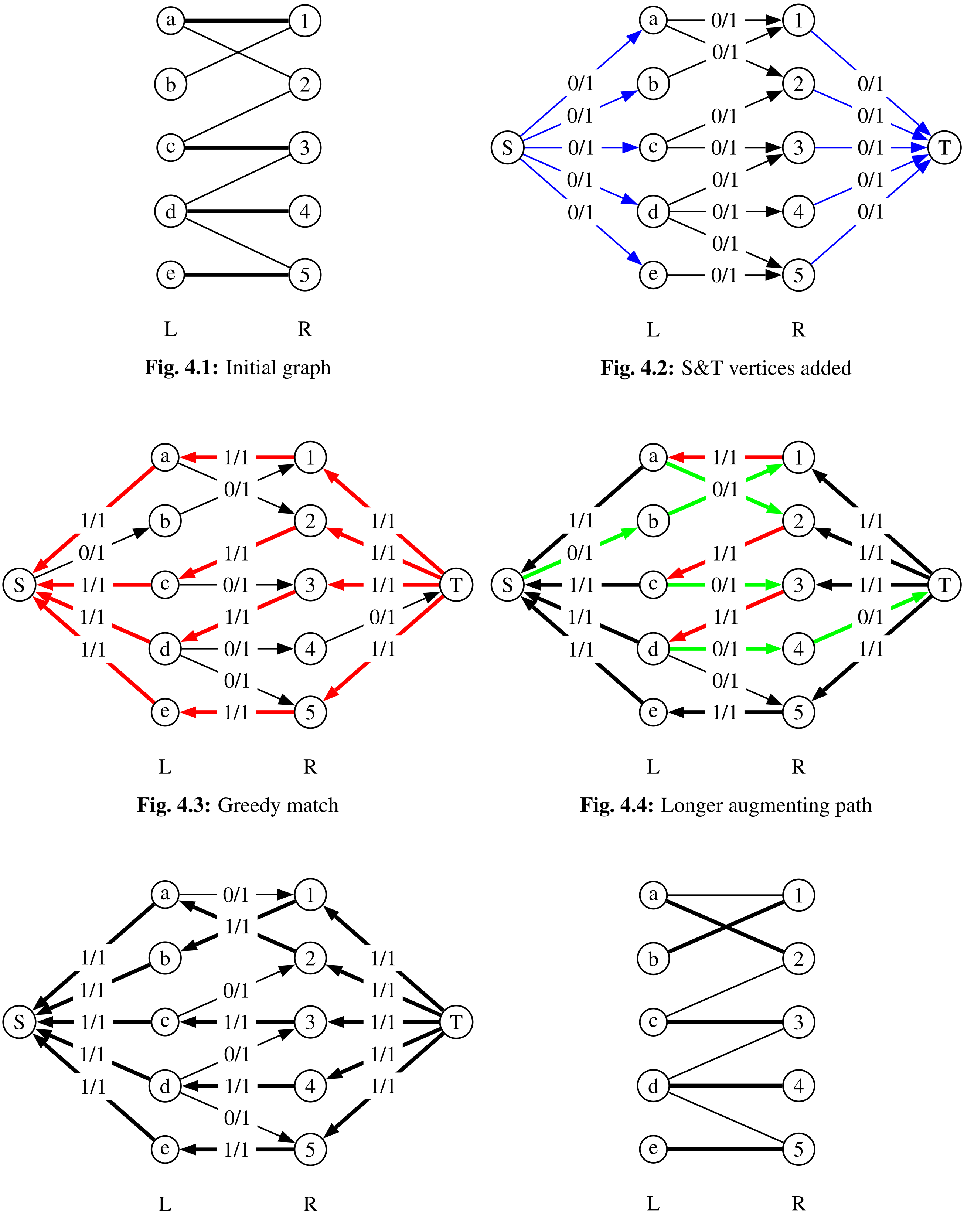


Fig. 4.1: Initial graph

Fig. 4.2: S&T vertices added

Fig. 4.3: Greedy match

Fig. 4.4: Longer augmenting path

Fig. 4.5: New matching

Fig. 4.6: Maximum matching

Fig. 4.7: The FordFulkerson algorithm [4]

Algorithm 1: The Ford Fulkerson algorithm

```
void $FordFulkerson$(Graph $G$)
begin
    Global int $capacity[][] \leftarrow getCapacity(G)$
    Global int $n \leftarrow |V(G)|$
    Global int $source \leftarrow 0$
    Global int $sink \leftarrow n - 1$
    Global $pred \leftarrow$ $new$ int$[n]$
    Global $visited \leftarrow$ $new$ boolean$[n]$
    Global $Q \leftarrow$ $new$ Queue$()$
    while $bfs(G)$ do
        $update()$

boolean $bfs$(Graph $G$)
begin
    $clear(Q)$
    $fill(pred, -1)$
    $fill(visited, false)$
    $enqueue(source, Q)$
    $visited[source] \leftarrow$ $true$
    while $(\neg isEmpty(Q))$ do
        int $v \leftarrow dequeue(Q)$
        if $v =$ $sink$ then return $true$
        for $w \leftarrow$ $0$ to $n$ do
            if $\neg visited[w]$ and $capacity[v][w] > 0$ then
                $pred[w] \leftarrow v$
                $enqueue(w, Q)$
                $visited[w] \leftarrow true$
    return $false$

void $update()$
begin
    int $f \leftarrow minCost()$
    int $v \leftarrow sink$
    while $pred[v] \neq -1$ do
        int $u \leftarrow pred[v]$
        $capacity[u][v] \leftarrow capacity[u][v] - f$
        $capacity[v][u] \leftarrow capacity[v][u] + f$
        $v \leftarrow u$

int $minCost()$
begin
    int $minCost \leftarrow \infty$
    int $v \leftarrow sink$
    while $pred[v] \neq -1$ do
        $minCost \leftarrow minimum(minCost, capacity[pred[v]][v])$
        $v \leftarrow pred[v]$
    return $minCost$
```

Ford Fulkerson's algorithm [4] for finding a maximum flow in a graph is shown in Algorithm 1. The *Ford Fulkerson* procedure on line 1 takes a graph G as an argument and represents the start of the algorithm. The graph passed as an argument holds information about the capacity of each edge. Line 3 stores the capacity of the graph G in a global two-dimensional integer array. The capacity of an edge between vertices u, v is stored in $capacity[u][v]$. Line 4 declares an integer n used to store the number of vertices in the given graph. Lines 5 and 6 declare two integers called $source$ and $sink$ representing references to the first and the last vertices in the given graph.

In line 7 a vertex-indexed array $pred$ is declared that is used to store references to the previous vertex discovered in the breadth-first search tree. Line 8 declares a vertex-indexed array $visited$ used to keep track if a vertex has already been visited in the breadth-first search procedure. Line 9 declares a queue of integers Q used to store vertices in the order they are visited.

The algorithm consists of repeated calls to the breadth-first search procedure (line 10). In line 11, there is a call to the update procedure to update the capacities of the edges after finding augmenting paths.

The *breadth-first search* procedure starts at line 12 and takes a graph G as an argument. In lines 14 to 16, the queue Q, $pred$ and $visited$ array are reset to their default values. The breadth-first search call always start from the first vertex, representing the source. This vertex is enqueued on queue Q in line 17 and is marked as visited in line 18.

Lines 19 to 26 contain a loop performing changes on the queue Q that is used to store the vertices that will make an augmenting path. The loop starts on line 20 by dequeuing a vertex from the queue Q and storing it in v. Line 21 forces the breadth-first search procedure to return $true$ when the last vertex in the graph (the sink) is reached. Lines 22 to 26 contain a loop over all the vertices w in the graph. In line 23 a check is performed to see if the discovered vertex w hasn't been already visited and it gets considered if it still has available capacity. Line 24 stores the index value of w's predecessor in the augmenting path. Line 25 enqueues the discovered vertex w on the queue Q and marks it as visited in line 26. Finally, the breadth-first search procedure returns *false* when no more augmenting paths are found.

The $update$ procedure at line 28 is used to update the capacities of the edges in the graph to reflect the updated flow. Line 30 declares an integer f used to store the amount of flow to be added along the found path. The value for f is calculated in the $minCost$ procedure at line 37. Line 31 declares an integer v, initially representing the sink which is the last vertex in the graph. Lines 32 to 35 contain a loop that updates the capacities of the edges in the augmenting path. Line 33 uses the integer u to store the index of a vertex that is the predecessor of vertex v along the path. Lines 34 and 35 update the flow along the edge from u to v and v to u by the found difference. Finally, v is updated on line 36 to take the value of u such that the while loop continues to change the capacities of the remaining edges along the path.

The $minCost$ procedure at line 37 is used to find the amount of flow to be sent along the new augmenting path from the sink to the source. The value is equal to the minimum capacity that is still available along the edges of the path. Line 39 declares an integer $minCost$ used to represent the current value of the minimum cost, initialized to a big value. Line 40 declares an integer variable v initially used to store a reference to the sink vertex. Lines 41 to 43 contain a loop over the edges along the path, starting from the sink toward the source. Line 42 updates the value of $minCost$ if the current edge in the loop has a smaller capacity left than what was previously discovered. The $minimum$ procedure returns the minimum of two given integers. Line 43 updates the value of integer v to its predecessor so the loop can continue along the path until it reaches the $source$. At the end of the $minCost$ procedure, the value of the currently found $minCost$ variable is returned.

4.2 Tarjan's algorithm

The next step in the all-different algorithm is to find the strongly connected components of the graph. In order to do this, we introduce now Tarjan's algorithm [14] for finding strongly connected components (SCCs) in a given graph G. A strongly connected component C of a directed graph G is a maximal set of vertices such that every vertex is reachable from every other vertex by following the directed edges.

The algorithm starts by visiting every vertex in the directed graph in a depth-first search manner. During the search, vertices are added to a stack in the order they are discovered only if they were not already part of the stack.

There are four kinds of edges in a depth-first traversal: tree edges, forward edges, back edges and cross edges. Tree edges are edges between a vertex and vertices that were not previously visited. Forward edges are from ancestor vertices to descendant vertices that were already visited during the search. Back edges link descendant vertices back to already visited ancestor vertices. Lastly, cross edges represent edges between two vertices that are neither an ancestor nor a descendant for each other, such as the edges between different components of the graph.

For each vertex in a graph G, the algorithm keeps track of two properties. The first property we remember is the $index$ of the vertex, its consecutive order of discovery during the depth-first search, a value that will remain constant throughout the algorithm. The second property associated with all the vertex is named low and keeps track of the lowest (oldest) ancestor reachable from their position in the graph. As each vertex of the graph gets discovered in a depth-first search manner, the initial value that low gets assigned is the same as the order the vertex was discovered. The value for low may get changed during the search if an ancestor for the vertex is discovered by following a back edge. After the recursive depth-first search finishes visiting all of a neighbours of a vertex, the vertex's value for low gets updated to the lowest index of its oldest reachable ancestor. If the value for low of a vertex stays the same even after visiting all of its neighbours, then it means it is the root of a strongly connected component or it's an individual vertex that will be a component on its own.

For the purpose of finding SCCs, every time a vertex is discovered in the search, the vertex is put on a stack S. After returning from the recursive DFS calls on all of the adjacent neighbours of vertex, if both $index$ and low have the same value, it means we've finished discovering a SCC. The vertices in the newly discovered strongly connected component will be popped from the stack S until we reach the current vertex representing the root of the SCC. Vertices that are part of other SCCs are still left on the stack as they have different roots represented by the value of low.

Fig. 4.8 shows a bipartite directed graph with 18 vertices. There are 9 backward edges that represent the maximum matching found using the Ford-Fulkerson algorithm [4] in Section 4.1. The rest of the possible edges are assigned a forward direction. This graph is similar to the graphs that will be generated to represent a row, column or one of the nine sub-grids of the Sudoku puzzle. For example, the left partition of the graph will contain the nine variables of a Sudoku row, and the right partition will contain the digits from 1 to 9. The edges will then link each variable of the row to values that are still available in their domain.

Table 4.1 contains the output after running the Tarjan's algorithm [14] on the graph in Fig. 4.8. As previously discussed, vertices are visited in a depth-first search manner, in this case starting from the first vertex. Each vertex has two properties associated to it: its order of discovery or $index$, and its oldest reachable ancestor. Initially, both properties take the value of the index, but as the algorithm progresses the values for low may become lower after visiting all of the neighbours of the vertex and finding the oldest (lowest) reachable ancestor. All vertices are pushed on a stack in the order they are visited. Once a back edge is found in the graph, vertices that are currently on the stack get popped including the root of the newly discovered SCC. Note that vertices may still remain on the stack as they will be part of different components that have other roots. As each vertex is made part of a component, its value for low gets updated to a large value to mark it as already part of a component.

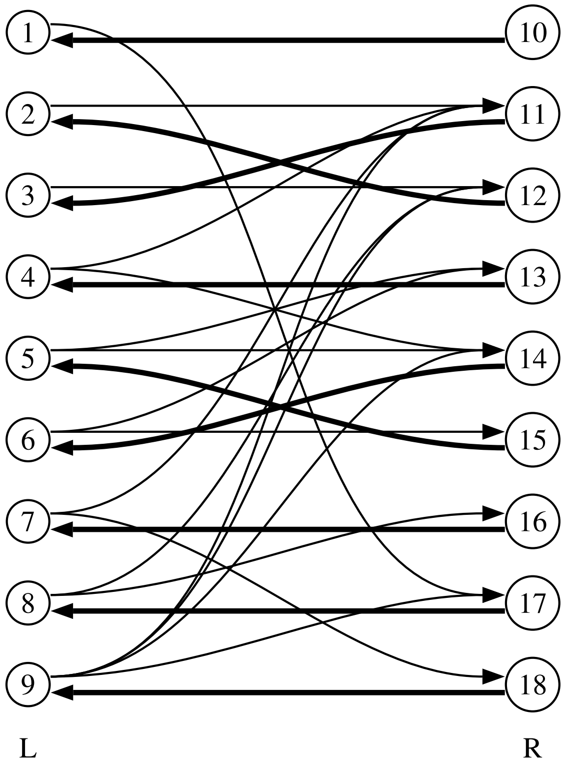


Fig. 4.8: An initial graph

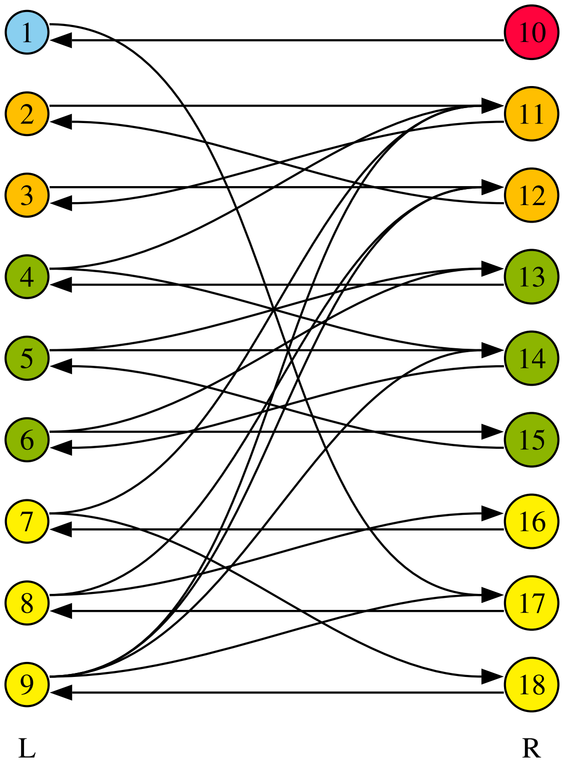


Fig. 4.9: The components of the graph

vertex	index	low	stack	new component	color
1	1	1	[1]		🟦
17	2	2	[1, 17]		🟨
8	3	2	[1, 17, 8]		🟨
12	4	4	[1, 17, 8, 12]		🟧
2	5	4	[1, 17, 8, 12, 2]		🟧
11	6	4	[1, 17, 8, 12, 2, 11]		🟧
3	7	4	[1, 17, 8, 12, 2, 11, 3]		🟧
			[1, 17, 8]	[3, 11, 2, 12]	
16	8	2	[1, 17, 8, 16]		🟨
7	9	2	[1, 17, 8, 16, 7]		🟨
18	10	2	[1, 17, 8, 16, 7, 18]		🟨
9	11	2	[1, 17, 8, 16, 7, 18, 9]		🟨
14	12	12	[1, 17, 8, 16, 7, 18, 9, 14]		🟩
6	13	12	[1, 17, 8, 16, 7, 18, 9, 14, 6]		🟩
13	14	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13]		🟩
4	15	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13, 4]		🟩
15	16	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13, 4, 15]		🟩
5	17	12	[1, 17, 8, 16, 7, 18, 9, 14, 6, 13, 4, 15, 5]		🟩
			[1, 17, 8, 16, 7, 18, 9]	[5, 15, 4, 13, 6, 14]	
			[1]	[9, 18, 7, 16, 8, 17]	
			[]	[1]	
10	18	18	[10]		🟥
			[]	[10]	

Table 4.1: The depth-first search performed in Tarjan's algorithm [14]

Algorithm 2: Tarjan's algorithm [14] for finding the Strongly Connected Components of a Graph

```
 1 void $Tarjan$(Graph $G$)
 2 begin
 3     Global int $index \leftarrow 0$
 4     Global int $components \leftarrow 0$
 5     Global int $n \leftarrow |V(G)|$
 6     Global $S \leftarrow$ $new$ Stack()
 7     Global $stacked \leftarrow$ $new$ boolean$[n]$
 8     Global $id \leftarrow$ $new$ int$[n]$
 9     Global $low \leftarrow$ $new$ int$[n]$
10     for $u \in V(G)$ do
11         if $\neg stacked[u]$ then $dfs(u, G)$

12 void $dfs$(int $u$, Graph $G$)
13 begin
14     $push(u, S)$
15     $stacked[u] \leftarrow true$
16     $low[u] \leftarrow index$
17     $index \leftarrow index + 1$
18     int $min \leftarrow low[u]$
19     for $v \in N(u, G)$ do
20         if $\neg stacked[v]$ then $dfs(v, G)$
21         $min \leftarrow minimum(low[v], min)$
22     $low[u] \leftarrow minimum(low[u], min)$
23     integer $v$
24     repeat
25         $v \leftarrow pop(S)$
26         $id[v] \leftarrow components$
27         $low[v] \leftarrow n$
28     until $v \neq u$
29     $components \leftarrow components + 1$
```

The graph in Fig. 4.9 shows the 5 strongly connected components identified after running Tarjans algorithm [14] on the initial graph in Fig. 4.8. Vertices that have the same background are part of the same component; meaning that one vertex can reach every other vertex in the component by following the directed edges. Note that some vertices make a component on their own as there is no way to reach them again in a cycle.

Tarjan's algorithm [14] for finding strongly connected components is shown in Algorithm 2. The *Tarjan* procedure in line 1 takes a Graph G as an argument and represents the start of the algorithm. The global integer $index$ declared in line 3 stores the current order in the depth-first search, and gets incremented each time a vertex is discovered. The integer $components$ in line 4 keeps track of how many components were identified in the given graph. The integer n in line 5 represents the number of vertices in the given graph.

In line 6 a declaration is made for a *Stack* data structure that will hold all the vertices that are part of the same strongly connected component. Line 7 contains a declaration for the $stacked$ vertex-indexed array of booleans used to keep track if a vertex is present or not on the stack S.

An integer vertex-indexed array id is declared in line 8 to store the id of the strongly connected component of each vertex. Line 9 declares a vertex-indexed array low used to store the topmost reachable vertex in the depth-first search tree through a back edge. Lines 10 and 11 contain a for loop that calls a procedure for performing depth first search on every vertex in the given graph that is not currently stacked.

Line 12 declares a recursive procedure for performing a depth-first search starting from vertex u in graph G. The procedure starts by pushing the vertex u on the stack S (line 14) and then updating the value in the $stacked$ array (line 15) to reflect the change. Line 16 sets the vertex's v value for low to $index$ as every vertex assumes to have no ancestors until a back edge to one is found. In line 17, the value for $index$ is incremented by one to keep count of the current number of discovered vertices. Line 18 declares an integer min used to hold information about the oldest, lowest ancestor that will be discovered later in the search and is initialized to take the same value of the current vertex.

Lines 19 to 21 loop through every vertex v that is a neighbour of vertex u. The procedure $N(u, G)$ on line 19 returns a list of neighbouring vertices of a given vertex, not including the given vertex. If the discovered neighbour v is not currently on the stack S, then we proceed to make a recursive call to the depth-first search function on the found vertex.

After the recursive call finishes processing the sub-tree of the neighbouring vertex v, min gets updated on line 21 to store the topmost reachable vertex that could be the same, or even higher in the tree if a higher back-edge from v was discovered.

Line 22 updates the topmost reachable vertex of vertex u to reflect any changes after processing all the neighbouring vertices. Line 23 declares an integer v used to store a vertex that for the repeat-until loop in lines 24 to 28. The loop begins by popping vertices out of the stack s on line 25. Each vertex v found on the stack is assigned the current component id. Line 26 assigns the last leaf vertex of the graph as a topmost reachable vertex of vertex v in order to mark it as already part of a component. The repeat-until loop in line 24 to 28 is runs until reaching the root of the component.

The depth-first search procedure finishes on line 29 by updating the total number of strongly connected components discovered.

4.3 Deletion of edges from the graph

The final step in the all-different algorithm uses the knowledge gained after computing a maximum matching and finding the strongly connected components in the graph. For this step, every edge that does not belong to any matching covering the vertices in the left partition of the graph will be deleted. The algorithm shown in Algorithm 3 is a slight adaptation of the fourth and final step in Jean-Charles Régin's paper [11] describing the all-different constraint. The algorithm starts by computing a matching using the Ford Fulkerson algorithm [4]. Having the match, a breadth-first search is performed on the free vertices in the graph and traversed edges are marked as $used$. Tarjan's algorithm [14] is then used to find the strongly connected components in the directed graph. Each edge connecting two vertices in the same component is marked as $used$. Finally, all the edges in the graph that are $unused$ and are not part of the matching will be deleted from the graph. The $unused$ edges represent edges that are not part of any alternating path of even length, and therefore belong to no possible matching in the value graph of the constraint. The removed edges in the graph correspond to values getting removed from the domains of the variables, and represents the way the search spaced for a solution is pruned.

18	23	23	245	456	456	279	378	23589

Fig. 4.10: A Sudoku row used to illustrate the all-different algorithm

Algorithm 3: Remove edges from the value graph of a constraint of difference

```
$RemoveEdgesFromGraph$(**Graph** $G$)
// *RemovedEdges* is the set of edges removed from G
// $M(G)$ is a matching over $G$ using the Ford-Fulkerson algorithm [4]
// The function returns *RemovedEdges*
begin
    Mark all the edges in graph G as $unused$
    Initialize *RemovedEdges* to the empty set
    Compute a mathing over G using the FordFulkerson algorithm [4]
    Perform a breadth-first search starting from the free vertices and mark all visited edges as $used$
    Compute the strongly connected components of graph G using Tarjan's algorithm [14]
    Mark each edge that connects two vertices in the same component as $used$
    for each edge $e$ in graph $G$ that is $unused$ do
        if $e \in M(G)$ then
            mark $e$ as $vital$
        else
            *RemovedEdges* ← *RemovedEdges* $\cup$ $\{e\}$
            Remove $e$ from $G$
    return *RemovedEdges*
```

The recently described last step of the all-different algorithm is the general implementation of the all-different constraint. The properties of the all-different constraints used in the Sudoku puzzle make the last step of the all-different algorithm much easier. It is good to notice that the chosen value graph used in this report has 18 vertices in two partitions. The Sudoku row shown in Fig. 4.10 corresponds to the value graph that was used in this report and was chosen as an example to illustrate the effects of running the all-different algorithm on the 9 variables involved in the constraint. The value graph of the constraint is shown in Fig. 4.11. The variables of the Sudoku row in the left L partition of the value graph have names from A to I. Each variable has edges to vertices in the right R partition of the graph representing digits in the domain of the variable. As all the constraints of difference in the Sudoku puzzle have 9 variables taking at most 9 possible values, the two partitions of the graph will always be balanced. This property makes the last step of the all-different constraint a straightforward process in the case of the Sudoku puzzle. This alternative algorithm for the last step in the all-different constraint considers only the edges that connect two vertices from different components of the value graph. Edges of interest are highlighted in Fig. 4.11. The direction of the edges indicate two outcomes and are coloured in blue and red as shown in Fig. 4.12. Forward edges (coloured in red) indicate that the digits from the right R partition should be removed from the domain of the variables in the L partition as they will never be part of an even alternating path and therefore cannot be part of a solution where the constraint holds. Backward edges (coloured in blue) indicate finding the unique assignment to a variable. In the case shown in Fig. 4.12, the first variable A of a selected Sudoku row will certainly take the value of 1 as all other values got removed from its domain.

Fig. 4.13 shows the final state of the Sudoku row after the last step in the all-different algorithm. Values coloured in red (underlined) correspond to edges that got removed from the value graph of the constraint. Values coloured in blue represent the unique value that is assigned to a variable as all other values from the domain got removed.

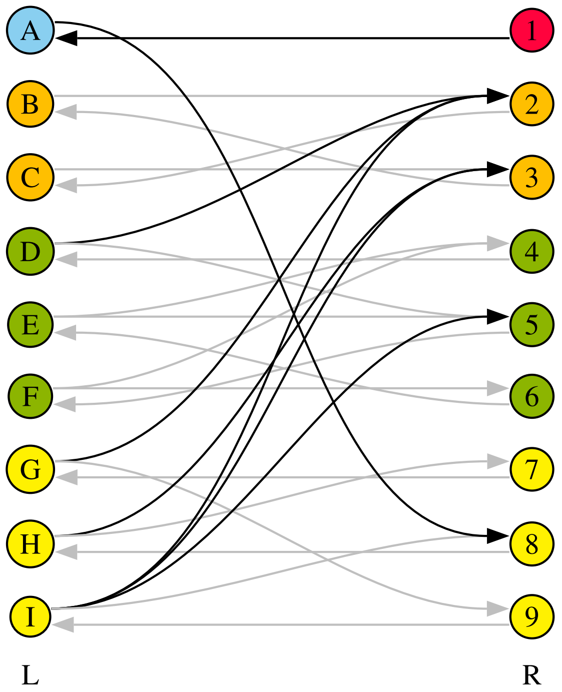


Fig. 4.11: Edges connecting vertices in separate components

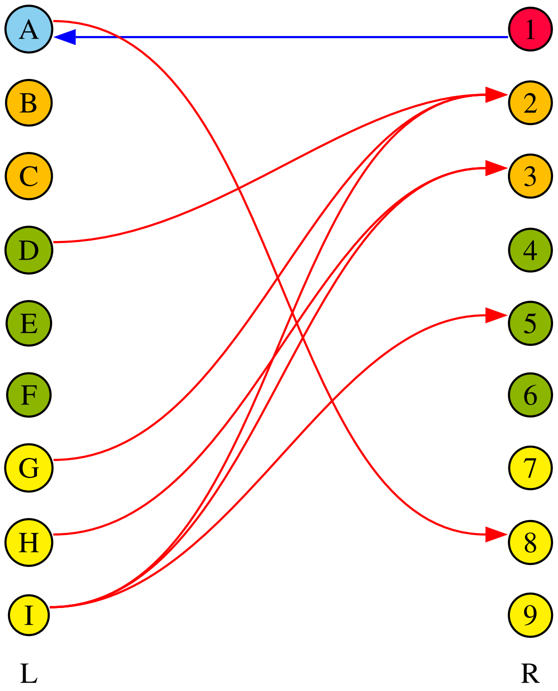


Fig. 4.12: Deletions (red, forward direction) and assignments (blue, backward direction) for the variables

18	23	23	245	456	456	279	378	23589

Fig. 4.13: Removal of values from the domain (in red, underlined), and assignments (in blue) in a Sudoku row

Notice how in the row in Fig. 4.13 it is certain that the first variable takes the value 1 as no other variables have it as an option. Another interesting thing to notice is how the second and third variables both have $\{2,3\}$ as their domain. The all-different algorithm works by spotting such Hall sets [5], where a Hall set is when n variables have their domain equal to a subset of a set of values D that has cardinality n. For each Hall set, one can remove the identified values from the domains of the neighbouring variables that were not part of the identified Hall set. For example, another Hall set in the mentioned row is between the 3 variables that have the domains $\{4,5\}$, $\{4,5,6\}$ and $\{4,5,6\}$, as 2 got removed by the previously identified Hall set. This means that there are 3 variables that together can take 3 values: $\{4,5,6\}$. Knowing this, the digits 4, 5 and 6 can be removed from the neighbouring variables as they will be consumed by the three variables in the Hall set, without knowing which variable will take which value.

Chapter 5

All-Different Demonstration

This chapter provides a demonstration of the Java program developed to solve Sudoku puzzles using Constraint Programming. The Graphical User Interface of the program is split into two parts, the left-hand side showing a Sudoku puzzle, while the right-hand side shows the value graph of an all-different constraint. Each time a row, column or 3×3 sub-square is selected by the user, the corresponding 9 variables are highlighted in the Sudoku puzzle. The steps in the all-different alogorithm perform changes on the graph of the selected constraint of difference.

Every Constraint Satisfaction Problem (CSP) has variables with domains associated to them and constraints that should hold for a solution. For the purpose of this project, there are 81 variables representing the 9×9 cells in the Sudoku grid, each with an initial domain of integers from 1 to 9 as shown in Fig. 5.1. The aim is to reduce the domain of all the variables to a single value, the correct value that is part of the solution. As all initial Sudoku puzzles start with a partially completed grid, values that represent clues are reflected as variables that have an initial domain of a single value, the known digit. Such variables keep their initial assignment throughout the algorithm, as no other values are present in their domain.

The constraints for a Sudoku puzzle can be expressed in a natural language as such: "digits may not appear twice in the same row, column or 3×3 sub-square". The rules of the puzzle highlight the need of thinking in terms of 9 rows, 9 columns and 9 sub-squares, each of them having digits from 1 to 9 with no repetition. The Sudoku puzzle will therefore be modeled as a CSP [12] that has 27 all-different constraints, one for each of the 9 rows, columns and sub-squares. Each constraint contains 9 variables that should take different values. As each variable is part of a row, column and sub-square, it is therefore involved in 3 constraints of difference at the same time.

One gets to the solution by running the all-different algorithm on rows, columns and sub-squares to remove values from the domain of the variables. Initially, the known values given at the start of the puzzle will be removed from the domains of possible values in the variables that are part of the same row, column or sub-square. As the algorithm progresses, it will eventually remove enough values from the domain of a variable until a single value is left, representing the correct answer for that position. Once a value is found, there is a waterfall-like effect across the puzzle as the knowledge is propagated across the grid. The propagation takes place across all the variables that are in the same row, column and sub-square with the one just solved. This keeps the 3 constraints of difference in a consistent state, as all the variables will have remaining domains not containing any already solved values. Initially, the propagation mechanism is disabled in the program such that all changes to the puzzle are made by the all-different algorithm that detects such inconsistencies and removes them. Propagation after assignments can be enabled from the Settings menu.

In order to test and show that the all-different algorithm is implemented correctly and performs the right decisions, a demonstration is hard coded consisting of running the all-different algorithm on a sequence of 5 rows, columns and sub-squares. The step by step demonstration can be run using the *Show demo* button and can be paused at any time. Alternatively, the same sequence can be followed by manually selecting the rows, columns and sub-squares mentioned in Appendix B.

The all-different algorithm consists of a number of operations performed on the value graph of a selected constraint of difference. All operations on the graph are shown in a step by step fashion in the right-hand side of the screen. First, a greedy match is performed on the graph to associate most of the variables with a value. The second step is to compute a maximum matching for the 9 variables in the left partition of the graph. The maximum matching is computed using the Ford Fulkerson algorithm [4] introduced in Section 4.1. Fig. 5.1 shows the maximum matching for the constraint of difference in the 1st column of the puzzle. The third step uses information from the maximum matching to find the strongly connected components in the value graph of the constraint, by using Tarjan's algorithm [14] introduced in Section 4.2. Fig. 5.2 shows vertices belonging to the same component having the same background colour. Finally, the edges connecting vertices that belong to different components get colored in red or blue based on their direction. Red edges connect a variable with a value that was removed from its domain during the algorithm. Blue edges highlight the single value a variable is forced to take, as it is the only option left in its domain. The state of the 1st column after running the all-different algorithm is shown in Fig. 5.2. Notice how the all-different algorithm performs changes only to the selected 9 variables inside a row, column or 3 × 3 sub-square.

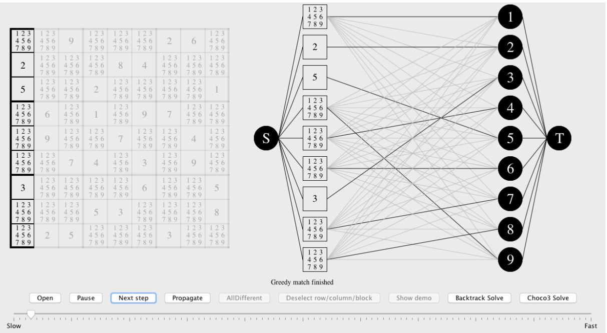


Fig. 5.1: 1st column after finding a maximum matching

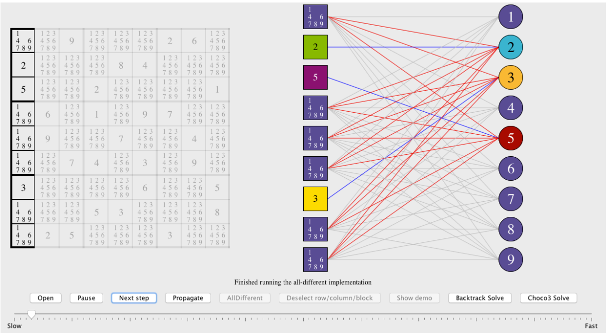


Fig. 5.2: 1st column after running the all-different algorithm

The same steps are followed for the 5th row (Fig. 5.3), the 6th row (Fig. 5.4) and the 4th 3 × 3 sub-grid (Fig. 5.5). As seen in the figures mentioned before, all the previous runs of the all-different algorithm removed values from the domain of the variables that represented clues in the initial puzzle. Notice how in Fig. 5.5 there are two variables in the 1st column of the puzzle, each having a domain of $\{1, 8\}$. One does not know which of the two variables will take which value, but since the two values will have to be used by the two variables, this means both values should be removed from the other variables in the same constraint. This detail is spotted by the all-different algorithm that decides to remove the two values from the domains of the neighbouring variables. The last selection in this demonstration calls the all-different algorithm on the 1st column of the puzzle once again. Fig. 5.6 shows the state of the 1st column after running the all-different algorithm the second time. Notice how only two variables in the 1st column have the values 1 or 8 in them, compared to last time. It is good to mention once again that the all-different algorithm does not make guesses, instead it looks for Hall sets [5] in the graph to remove values from the domains of the variable that will never be part of a solution.

The algorithm is intelligent in a sense that it mimics the human strategy when solving a Sudoku puzzle. At some stages of the game there is enough information to know that, for example, two particular values will fill two cells, but no information about which value corresponds to which cell. The hardness of a Sudoku puzzle increases with the number of such dilemmas, and the number of variables having the same values in their domain [6]. By using the all-different algorithm, one can ensure the progress towards a solution, despite not knowing which value goes where. It is good to note that although humans may choose a guessing approach to continue towards a solution, the all-different algorithm performs no guessing and works only with the information about the current state of the puzzle.

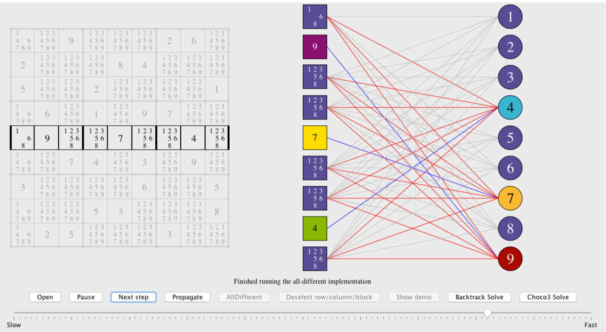


Fig. 5.3: 5th row after running the all-different algorithm

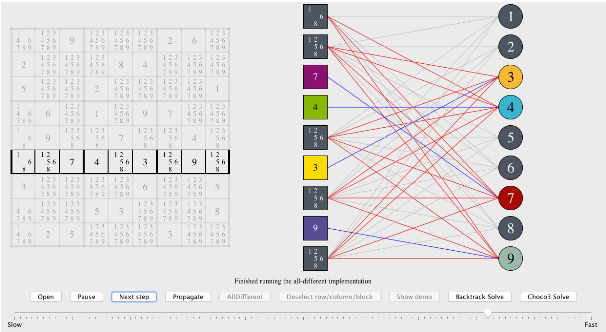


Fig. 5.4: 6th row after running the all-different algorithm

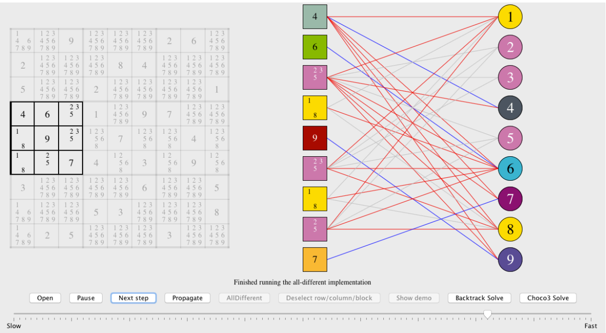


Fig. 5.5: 4th 3 × 3 sub-grid after running the all-different algorithm

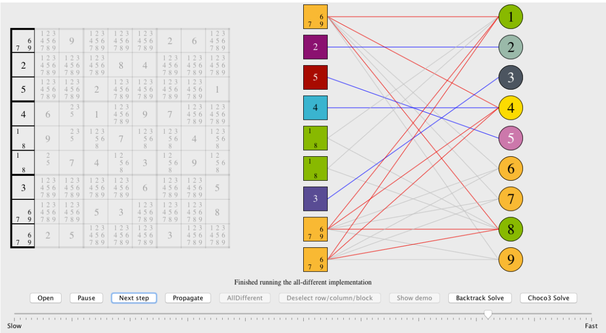
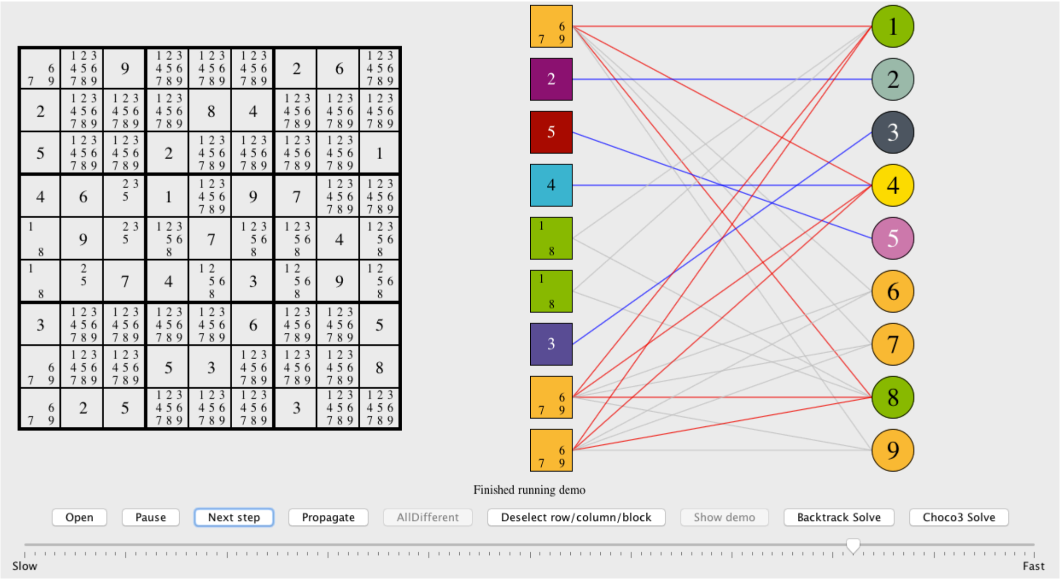


Fig. 5.6: 1st column after running the all-different algorithm



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

Finished running demo

Fig. 5.7: State of the program after running the demo

The end result of following the 5 steps in the demonstration can be seen in Fig. 5.7. In the 1st column there are two cells with a domain of 1, 8. Although, one does not know yet which cell takes which value, the human and the all-different algorithm is sure that the digits 1, 8 will be distributed to the two particular cells. The last step illustrates the all-different algorithm decision to remove the values 1, 8 from the domains of the rest of the cells in the selection.

Chapter 6

Implementation

This section describes the program that was made to showcase how the all-different algorithm works. The programming language used throughout the project is *Java*, a popular language that supports Object Oriented Programming. One of the most important benefits of using Java for this project is that it is platform-independent, meaning that the same source code can be compiled to run on different processors and Operating Systems, allowing for greater portability. Java's standard library includes support for concurrency, building the Graphical User Interface (using the *Swing* library), and various data structures that are needed in the project.

Two classes *Var* (variable) and *Constraint* are used to model the Sudoku problem as a Constraint Satisfaction Problem. Each of the cell in the Sudoku puzzle gets represented as a variable that has an initial domain set to a single digit (in the case of the clues present in the initial puzzle) or an initial domain of digits from 1 to 9 for blanks in the original puzzle. The *Constraint* class is used to specify that variables in the same row, column or sub-square should be all different. The constraints are used throught the program to check if they still hold in a solution or it they are consistent with each other during the search for a solution.

From a high-level view, the Model-View-Controller (MVC) design pattern was chosen for the program's arhitecture. By using a MVC arhitecture one can achieve a clean separation of concerns, where the Model object encapsulates the data and defines the operations that can done on that data, while the View object reads the data from the Model and presents it to the User. Controller objects are used to link the Model to the View by communicating changes such as User input that update the state of the Model and the View. In practice, most of the time the View and Controller roles are combined as an object that is responsible for rendering the User Interface and handling events such as mouse clicks.

The program reads partially completed Sudoku puzzles from *.txt* files that can be opened by choosing the option from the File menu. Files should contain 9 rows filled with digits from 1 to 9, separated by spaces, representing clues in the initial puzzle. Unknown values in the initial puzzle that have to be filled by the player are represented in the file by the 0 digit. By default, the program tries to load the *lockedset.txt* file used in the demonstration that is stored in the *puzzles* directory inside the current working directory. Alternatively, if the default file is not found, the User should choose a file from his hard drive to open.

The Graphical User Interface (GUI) is inside a *JLayeredPane* which is a special *Swing* component that allows components inside it to overlap according to their specified depth. In this program, the backmost layer contains a *JPanel* responsible for the drawn edges in the graph. The layer above contains the rest of the GUI represented by the Sudoku puzzle on the left-hand side of the screen, and the vertices of the graph on the right-hand side of the screen. By using a layered panel, the edges of the graph can be drawn behind the vertices.

The Sudoku Grid is rendered on the left-hand side of the screen and is made up of 9×9 Sudoku Cells positioned at specific locations. Each Sudoku Cell is aware of its position in the grid and has a square *JPanel* to textually render the current domain of a variable to visualize the markup of the puzzle [3]. The border of the *JPanel* has different widths according to the position of the variable in the grid.

Not visible to the human eye, a new Sudoku grid instance is drawn on top of the existing one, this time with all borders set to a width of 1px. The Sudoku Cells inside this second grid contain the same information about the state of the puzzle retrieved from the Model. Additional information such as a custom background will be applied to these cells as the algorithm progresses. Cells inside the second grid become visible once the user makes a selection. Once visible, the cells inside the selection are animated into their position inside the graph on the right-hand side of the screen. When the user deselects his choice, the nice cells are animated back into their original position in the grid and become invisible again.

Small adjustments had to be made to the size and alignment of the components, and the size of the fonts such that the Graphical User Interface would look as intended across different Operating Systems. Although Java is platform-independent, meaning that the same code will run accross platforms supporting Java, challenges in rendering the User Interface remain as each Operating System specifies its implementation for the specified font family. For example, Windows renders Serif fonts smaller than Mac OS does.

Digits in each cell of the Sudoku grid are printed as graphics using the *drawString* method from the Graphics2D object associated to all GUI components. This approach was used to allow for a more precise alignment of the characters having different widths, as opposed to separating them using spaces. The borders in the Sudoku grid, and edges in the value graph of a constraint were made using the *drawLine* method of the Graphics2D object. The circles representing vertices in the right partition of the graph were made using the *fillOval* method.

The all-different algorithm is implemented as a method in the Model. The algorithm consists of multiple operations done on a bi-partite directed *Graph* representing the value graph of a constraint of difference. For the purpose of this project, the graphs and the operations on the graphs were programmed in a procedural way, as this allows for fewer and more readable lines of code. Graphs are represented as a $v \times v$ adjacency matrix (where v is the number of vertices in the Graph) filled with 0 and 1 to reflect if an edge exists between two vertices. Note that the adjacency matrix is not necessary symmetric, which allows specifing the direction of the edges. The graph is built with information that is read from the Model about the currently selected row, column or 3×3 sub-square. First, the Ford-Fulkerson algorithm [4] is used to compute a maximum matching over the 9 selected variables. Note that the Ford-Fulkerson algorithm uses two extra vertices called the $source$ and the $sink$ connected to the partitions of the graph with edges that have to be specified in the adjacency matrix of the graph. The Ford-Fulkerson algorithm is implemented according to the pseudo-code specified in Algorithm 1. The second step is to compute the Strongly Connected Components in the graph using Tarjan's algorithm [14] shown in Algorithm 2. Note that this step no longer uses the $source$ and $sink$ vertices from the previous step and changes should be made to the adjacency matrix to reflect this. Finally, only the edges that have a forward direction connecting a variable with a value from separate components are considered. Based on this final state of the graph, the values will be removed from the domain of the variables their are linked to. Each time a change occurs in the graph, the Model informs the View Controller that it needs to refresh to reflect the new changes stored in the Model.

The propagation mechanism used in this project represents the AC-3 algorithm developed by Alan Mackworth [9] in 1977. This 3rd revision of the Arc Consistency algorithm is more efficient than previous ones and one of the most often used because of its simplicity. The algorithm checks that all the constraints in the problem are consistent by considering all the pairs of variables in each constraint. When an inconsistency is found, such as when a variable still has a value in its domain that is already assigned to another varible, it gets removed from the first variable. Such inconsistencies occur after running the all-different algorithm, since the all-different algorithm only performs changes on the 9 variables of a row (for example) resulting in the need to update the intersecting columns and sub-squares to reflect the changes made. Each time a value gets removed from the domain of a variable, the algorithm rechecks the other constraints the variable is part of to see if they are still consistent after the removal. The algorithm is guranteed to finish after no more removal of values occur and all the constraints are checked. This algorithm is efficient in the sense that changes to the domains of a variable trigger a waterfall-like checks only on the variables that have a constraining relationship with the first one, as opposed to rechecking all the variables and the constraints of the problem.

The user selects which row, column or sub-square to solve next by clicking the first variable of the row/column, or the center of the sub-square. A selection triggers the movement of the row, column or sub-square to the right-hand side of the screen to be made part of the graph, all in an animated fashion. The selection is highlighted in the Sudoku puzzle, while the rest of the grid is faded out to let the user know that the all-different algorithm performs changes only on the 9 variables that are part of his selection. *Timer* objects are used to schedule repeated actions such as the fading out of the components or movement of the Sudoku cells. The Model contains a two-dimensional *ArrayList* of *Timer* objects that acts as a queue of animations. Timers on the same level start running at the same time until they all finish. Once all the timers on a level finish, the next level in the two-dimensional *ArrayList* is started, allowing us to queue a movement of a row after all the cells finish fading out. The user can deselect his choice before or after running the all-different algorithm, resulting in an animation that returns the selection to the original position in the Sudoku grid and fading in the puzzle so it is visible again.

The program includes a button used to play a demonstration of the all-different algorithm. The button calls the selection method, the all-different method and the deselection method for 5 rows, columns and sub-squares in a sequential way. The demonstration runs only if there is currently no selection made and the loaded Sudoku puzzle is the *lockedset.txt* file.

The pause/play mechanism used throughout the program to control the flow of the animations works by pausing the logic thread. The implementation consists of specifying multiple checkpoints at relevant steps in the animation where the program checks if the user requested that the animation should be paused. A synchronized $Object$ acts as a *lock* throught the program. The *wait* method is called on the $lock$ whenever the user requests a pause in the animation. The *notify* method is called to resume the animation from where it was paused. As the program uses *Timer* objects to perform repetitive tasks, all the timers will be stopped and started according to the user's needs by calling the methods *stop* and *start*.

The lower part of the User Interface includes a *JSlider*, which is a slider that can be used by the User to adjust the speed of the animation. The slider represents a bounded interval of speeds ranging from slow to fast. The speed of the animation is controled by specifing how much the logic thread should sleep when solving the problem and by what distance components should move during the animation. Note that only the logic thread is put to sleep and not the thread responsible for the program's interface, as this would freeze the Graphical User Interface and make it unresponsive.

Particular attention was given to make the software thread safe. It is crucial that the program's logic does not make the Graphical User Interface unresponsive. To achieve this, the program will run on two threads, one main thread responsible with all the processing need to be done by the program, and the Event Dispatching Thread. The Event Dispatching Thread (EDT) is a background thread used to invoke Swing methods that change the GUI and listen to events associated to components. Most Swing components are not thread safe, therefore bugs such as race conditions could unexpectedly arise if it was to perform the actions from normal threads. The EDT acts as a queue of events that are performed sequentially. The *Timer* objects used throughout this project to perform animations send events to the specified listeners making it crucial that the EDT does not get blocked.

A backtracking algorithm was implemented to solve the Sudoku puzzle by choosing values from the domains of the variables in the Sudoku puzzle. The algorithm goes through each available value in the domain of a variable, assigns it as its guess and calls the propagate method. If the value selected breaks a constraint, the propagate method will report this inconsistency that triggers the backtracking from that point. This method is slow as it does not use the all-different algorithm to prune the search space of the problem of such values that break the constraints. Alternatively, the user can solve the puzzle using the open-source *Choco3* library [10] that is used to model Constraint Satisfaction Problems in Java. The implementation models the currently loaded Sudoku and solves it using the same algorithms used in this project to provide proof that the solution found is the correct one.

Chapter 7

Conclusion

7.1 Summary

The aim of this project was to develop a Java program to be used during Constraint Programming lectures to introduce students to the all-different algorithm. This report started by introducing general concepts from Constraint Programming, a programming paradigm that originated from Artificial Intelligence. A chapter is dedicated to reviewing Jean-Charles Régin's paper [11] from 1994 describing the all-different constraint. The algorithm is a sequence of steps that perform operations such as finding a maximum matching and the strongly connected components in the value graph of a constraint of difference. The Ford-Fulkerson algorithm [4] was used in this project to compute the maximum matching, and Tarjan's algorithm [14] to find the strongly connected components in the graph. The Sudoku puzzle was chosen to illustrate the steps in the all-different algorithm. A Java program was developed to model the problem and provide a Graphical User Interface showing the puzzle and the value graphs of the all-different constraints. A description of the algorithms used and how the project was implemented in Java is provided in the paper.

7.2 Future Work

The program developed in this project implements a model for the popular Sudoku puzzle that has a 9×9 grid. There are multiple techniques and algorithms to solve such a puzzle in approximately the same time using a regular CPU. It is therefore difficult to accurately measure the performance of the algorithms since searching for a solution takes only hundreds of milliseconds. The benefits of using Constraint Programming to solve such problems, using techniques such as the all-different algorithm presented in this report, starts to become observable when trying to solve a bigger Sudoku grid, such as one of size 36×36. The performance in such a case is a matter of seconds, compared to hours or even days as shown in the experiments made by Stergiou and Walsh [13] who approached the Quasigroup completition problem [6], a problem that uses the same constraints as the Sudoku puzzle. Future work could mean developing models and a Graphical User Interface for general $n^2 \times n^2$ sized boards, along with a way to compare the performance of different algorithms.

Another interesting avenue for future work is to implement and make use of different heuristics that may prove to increase the performance of the program when trying to find the solution. The project uses human judgements as its heuristic, as one has to select which row, column or 3×3 sub-grid to solve next. An example of such a heuristic is one that will check which selection has variables with many values in their domains, as there may have values that are more likely to fail by breaking a constraint.

7.3 Personal Reflection

As a Joint Honours student who studies Computing Science and Business Management, I have come to appreciate the importance of an efficient algorithm over an expensive hardware upgrade. Powerful algorithms such as the one in this report can offer a great competitive advantage in business environments that deal with hard combinatorial problems on a daily basis. Such an example would be the scheduling of the deliveries that need to be made the next day, where an optimal solution should be found in a matter of hours. In the end I would like to quote Robert Tarjan [15], winner of a Turing Award, on the characteristics of a good algorithm: “It is one thing to have an algorithm that is theoretically good. It is another to have an algorithm that is simple enough that someone would want to program it and use it in practice”.

7.4 Acknowledgements

I would like to thank my supervisor, Dr. Patrick Prosser, and Ciaran McCreesh, for all their guidance and feedback provided throughout this project.

Appendices

Appendix A

Running the Program

Running the program from the command line is done after compiling the source code as follows:

```
> javac *.java
> java Sudoku
```

This will open the program loaded with the Sudoku puzzle */puzzles/lockedset.txt*.

Detailed debug output in the console can be activated using:

```
> java Sudoku debug
```

NOTE that the Choco3 library [10] found in *lib/choco-solver-3.3.1-with-dependencies.jar* has to be added to the CLASSPATH environment variable as follows:

```
      > export CLASSPATH=$CLASSPATH:/path/to/the/lib/choco-solver-3.3.1-
with-dependencies.jar
```

Alternatively, one could simply run the program from the *.jar* file that contains all necessary libraries as follows:

```
> java -jar sudoku.jar
```

Appendix B

Proof of concept

The following sequence of steps provide a visual proof of the all-different algorithm. The following steps require the user to run the all-different algorithm on 5 predetermined rows, columns or 3×3 sub-grids. For demonstration purposes, the Sudoku instance used is $/puzzles/lockedset.txt$.

The sequence is as follows:

```
> Open lockedset.txt Sudoku instance
>
> Select the 1st column
> Run the all-different algorithm
> Deselect the column
>
> Select the 5th row
> Run the all-different algorithm
> Deselect the row
>
> Select the 6th row
> Run the all-different algorithm
> Deselect the row
>
> Select the 4th 3x3 sub-grid
> Run the all-different algorithm
> Deselect the sub-grid
>
> Select the 1st column
> Run the all-different algorithm
> Deselect the 1st column
```

Alternatively, the user can press the $ShowDemo$ button to run the same steps.

Detailed description about why the last state of the program is a proof that the algorithm works as expected is provided in Chapter 5.

Bibliography

[1] Claude Berge. “Two theorems in graph theory”. In: *Proceedings of the National Academy of Sciences* 43.9 (1957), pp. 842–844.

[2] Charles J Colbourn. “The complexity of completing partial Latin squares”. In: *Discrete Applied Mathematics* 8.1 (1984), pp. 25–30.

[3] JF Crook. “A pencil-and-paper algorithm for solving Sudoku puzzles”. In: *Notices of the AMS* 56.4 (2009), pp. 460–468.

[4] Lester R Ford and Delbert R Fulkerson. “Maximal flow through a network”. In: *Canadian journal of Mathematics* 8.3 (1956), pp. 399–404.

[5] Ian P Gent, Ian Miguel, and Peter Nightingale. “Generalised arc consistency for the alldifferent constraint: An empirical survey”. In: *Artificial Intelligence* 172.18 (2008), pp. 1973–2000.

[6] Carla Gomes and David Shmoys. “Completing quasigroups or latin squares: A structured graph coloring problem”. In: *proceedings of the Computational Symposium on Graph Coloring and Generalizations*. 2002, pp. 22–39.

[7] Robert M Haralick and Gordon L Elliott. “Increasing tree search efficiency for constraint satisfaction problems”. In: *Artificial intelligence* 14.3 (1980), pp. 263–313.

[8] John E Hopcroft and Richard M Karp. “An nˆ5/2 algorithm for maximum matchings in bipartite graphs”. In: *SIAM Journal on computing* 2.4 (1973), pp. 225–231.

[9] Alan K Mackworth. “Consistency in networks of relations”. In: *Artificial intelligence* 8.1 (1977), pp. 99–118.

[10] Charles Prud’homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC INRIA Rennes LINA CNRS UMR 6241 COSLING S.A.S. 2015. URL: `http://www.choco-solver.org`.

[11] Jean-Charles Régin. “A filtering algorithm for constraints of difference in CSPs”. In: *AAAI*. Vol. 94. 1994, pp. 362–367.

[12] Helmut Simonis. “Sudoku as a constraint problem”. In: *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*. Vol. 12. Citeseer. 2005, pp. 13–27.

[13] Kostas Stergiou and Toby Walsh. “The difference all-difference makes”. In: *IJCAI*. Vol. 99. 1999, pp. 414–419.

[14] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[15] Robert Tarjan. *HP Labs - Inventor interview - Robert Tarjan : The art of the algorithm*. 2004. URL: `http://www.hpl.hp.com/news/2004/oct_dec/tarjan.html` (visited on 03/14/2016).