German University in Cairo
Faculty of Media Engineering and Technology
Computer Science Department

Ulm University
Institute of Software Engineering and Compiler Construction

Visualization of Grid-based and Fundamental CHR Algorithms

Bachelor Thesis

Author:	Arwa Ismail
Supervisor:	Prof. Thom Frühwirth
Co-supervisor:	Amira Zaki
Submission Date:	1 July 2012

This is to certify that:

(i) The thesis comprises only my original work toward the Bachelor Degree.

(ii) Due acknowledgment has been made in the text to all other material used.

Arwa Ismail
1 July, 2012

Acknowledgments

This thesis would have not been completed without:

- The support and encouragement of my family and their belief in me.
- Prof. Thom Frühwirth and Amira Zaki for their supervision and support. They are the main reason I was able to present a thesis that made me proud of myself.
- My friends here in Ulm. We've stuck through a lot together and managed to go through it all till the end.
- My friends in Egypt who have always stayed in contact even though we are miles apart. Their support has been undeniable and of great help.

Abstract

Visualisation or animation of a certain topic is an attractive method that can aid one's understanding of the concept behind such topic. This can be beneficial in the programming world, where for a certain implementation of an algorithm, one can follow its flow through its graphical representation. For this work, two fundamental algorithms as well as three grid-based ones are visualised through an implementation written using CHR with Prolog acting as a host language. The graphical aspect of the implementations is achieved using Prolog's tool kit XPCE.

Contents

List of Figures

Chapter 1

Introduction

Generally, the human brain is capable of imagining scenarios shown through images in motion, better than being presented only with written text or static pictures. A well-animated and easy-to-follow representation of an algorithm can aid one's understanding of such algorithm, since a static drawing or typed piece of code is not sufficient to describe its behaviour. There have been multiple works, for example [4] and [5], interested in such a concept. Usually, the idea is to employ a programming language with graphical capabilities, in some cases more than one language, to animate algorithms. In this work, the main focus is on visualising algorithms written in Constraint Handling Rules (CHR).

CHR is a high-level language based on logic and applying constraints through different kinds of rules as will be explained later in more detail through section 2.1. When introducing CHR to students, it is very important that they understand how the rules work and which commands are being performed, based on the choices made according to applying the constraints. For someone with no previous knowledge of such concept, presenting a visualisation of a certain CHR program can aid its comprehension and that is the aim of this thesis. This is done through visualising five chosen CHR algorithms as explained further in chapter 4.

In 1991, Thom Frühwirth introduced the idea of CHR through [6]. Then later on in his book[7], the concrete syntax of CHR was used with Prolog, acting as a host language, to officially introduce CHR's programming capabilities. One of the aims of this work is to follow the same approach of using the advanced and stable Prolog implementation of CHR from K.U. Leuven, and exploit it to visualise the algorithms. However, since Prolog on its own does not own any graphical capabilities[8], different possibilities of producing such visualisations are investigated and a suitable approach, that helps maintain the stability of the implementation for this work, is chosen. The five visualised algorithms are:

1. Finding the minimum of a set of given numbers
2. Wolfram's Cellular Automaton: Rule 110
3. Calculating the N^{th} value of the Fibonacci series
4. Solving the N-Queens problem
5. Finding the solution of a Sudoku puzzle

This thesis is divided into five chapters starting with this introduction. Chapter 2 discusses previous works that have influenced parts of this work, as well as an introduction to CHR and an overview on how each of the five algorithms works. In chapter 3, different graphical approaches that can be considered for Prolog are explored and presented, along with the choice made of the one to be used for the implementation part of this work. Also, the approach of the graphical representation of each algorithm is discussed on the same chapter. As for chapter 4, using the chosen implementation approach, each algorithm's implementation is explained in more detail as well as mentioning some related work. Finally, chapter 5 concludes the work that has been done throughout the thesis in addition to presenting ideas for future work.

Chapter 2

Background

This chapter is divided into three sections covering the researched material required for preparation purposes of this work. First, CHR is introduced alongside its three types of rules. Second, a section defines each of the five algorithms to be visualised, either mathematically or conceptually. Finally, a literature section summing up any influential or related works to this thesis's topic and interest.

2.1 Constraint Handling Rules

CHR is a concurrent constraint-based programming language which one can employ to impart user-defined constraints to a host language. It was originally embedded in Prolog, hence it is the most recommended and commonly-used implementation of CHR so far[7]. However, a wide range of languages today also support CHR implementations, such as Haskell, Java and C. Generally, for any CHR program, it is possible to mix the CHR code with the host language's statements.

Syntax and Semantics

A CHR constraint is represented in the same way a Prolog predicate is, i.e through a functor/arity pair where functor is the constraint name and arity is the number of arguments it carries. In order to introduce CHR and how such constraints are employed, we first introduce the syntax of the three types of rules that CHR offers[9]:

1. Simplification rules:
$$h_1, ..., h_i \iff g_1, ...g_j \mid b_1, ...b_k \tag{2.1}$$
2. Propagation rules:
$$h_1, ..., h_i \implies g_1, ...g_j \mid b_1, ...b_k \tag{2.2}$$
3. Simpagation rules:
$$h_1, ..., h_l \setminus h_{l+1}, ...h_i \iff g_1, ...g_j \mid b_1, ...b_k \tag{2.3}$$

A CHR rule consists of a set of *head* constraints $h_1, ..., h_i$, an optional set of *guard*s $g_1, ...g_j$, covering any checks that need to be made, and a set of *body* constraints $b_1, ...b_k$, where $i, j, k, l \geq 1$.

During execution, whenever a constraint is encountered it becomes *active*. A constraint is *matched* if it is found to be an instance of the *head* of a rule. Next, if the *head*(s) of the rule is(are) *matched* and a *guard* is present, then the rule is applied if the *guard* is met. Such rule is said to be *fired*. Otherwise if the constraint is not matched for this rule, the *active* constraint tries the next *head* matching. If no rule is applicable for the current constraint, then it becomes *passive*. A *passive* constraint is thrown into the *constraint store* which is a data structure for holding constraints throughout the CHR program execution.

The *head* constraints of a propagation rule as well as the *head* constraints $h_1, ..., h_l$ of a simpagation rule are called *kept* constraints. On the other hand, the *head* constraints of a simplification rule and the *head* constraints $h_{l+1}, ..., h_i$ of a simpagation rule are called *removed* constraints. Such naming was chosen because, for example, when a simplification rule is *fired*, its *head* constraints are *removed* from the constraint store. Whereas, when a propagation rule is *fired*, the *head* constraints are still *kept* in the constraint store. The same is applied to the *kept* and *removed head* constraints of a simpagation rule when it is *fired*. This will be explained further through the following example.

```
:- use_module(library(chr)).
:- chr_constraint cat/1, pet_owner/2, cat_owner/2.

simplification @ cat(X) , pet_owner(X,Y) <=> cat_owner(X,Y).

propagation @ cat(X) , pet_owner(X,Y) ==> cat_owner(X,Y).

simpagation @ pet_owner(X,Y) \ cat(Z) <=> X=Z | cat_owner(X,Y).
```

In order to write a CHR program, the first line must be added before any CHR code to load the CHR library. Also, in order to use any constraint, it must be declared first using the `chr_constraint` keyword as in the second line. For this short CHR program, there are three user-defined constraints; `cat/1`, `pet_owner/2` and `cat_owner/2`, where in a logical sense, a `pet_owner` can be, but not necessarily, a `cat_owner`. The labelling of the rules, "`simplification`" , "`propagation`" and "`simpagation`", are optional and are added to this example to aid referencing the different rules. Such labelling can also be used as an identity of its rule.

1. **Simplification**

   The first rule is a simplification rule where querying the two-part head simplifies into the constraint `cat_owner`; where a `pet_owner Y` with a pet `X` that is also a `cat`, is a `cat_owner`. When a simplification rule is *fired*, the constraints of the *head* are *removed* from the constraint store and replaced by the newly-thrown constraint(s) through executing the *body* of the rule, as previously explained in 2.1.

2. **Propagation**

   This type works similar to the simplification rule except that here, as mentioned before in 2.2, the constraints of the *head* are *kept* in the *constraint store* as well as

throwing in new ones. This makes it possible to compute all different combinations of an owner having different cats or a cat having more than one owner, e.g a cat that belongs to a family means that it belongs to each member of it.

3. **Simpagation**

   A simpagation rule is a mixture between the other two rules. The last rule is a simpagation rule that corresponds to the case of a pet owner having one or more cats. The rule is only fired when there is a `cat Z` and a `pet_owner X` with a pet `Y` and the condition `X=Z` is satisfied. Then, The constraints are handled as illustrated before in 2.3.

2.2 Algorithms

2.2.1 Minimum of a Set of Numbers

Given a set of numbers, the algorithm's purpose is to find the minimum number among them. The idea is to compare each number with the current minimum-so-far. Initially, the first number is set as the minimum since no other is there to compare with yet. The first two numbers are compared together and whichever is less than the other, is marked as the minimum-so-far. Then, that number is compared with the third input number and so on. The algorithm terminates whenever all of the input numbers have been tested and compared with the marked number. Thus, the minimum number of the set will eventually be marked. An abstract idea of how the algorithm code looks like is shown through Algorithm 1.

Algorithm 1 Minimum of a set

```
i ← 0
min_so_far ← set[i]
i ← i + +
while i < set.length() do
    if set[i] < min_so_far then
        min_so_far ← set[i]
    end if
    i ← i + +
end while
```

Note that *set* is an array of numbers the algorithm is performed on and *min_so_far* is the number marked to be the minimum.

2.2.2 Fibonacci: Top-Down Approach with Tabling

A Fibonacci sequence is a series of numbers where every subsequent number is the sum of the two numbers preceding it. The two base cases are 0 and 1 where the 0th and 1st element of a Fibonacci sequence is 1. For a number *N*, the top-down approach to calculate the `N`th element of the Fibonacci sequence starts with calculating the `I`th and `J`th element of the Fibonacci sequence, where `I` is `N-1` and `J` is `N-2`, then summing them up. For a better understanding of the derivation of such a sequence, the following mathematical relation defines it:

$$F_n = F_{n-1} + F_{n-2} \tag{2.4}$$

where the bases cases are defined as

$$F_0 = 1 \; and \; F_1 = 1 \tag{2.5}$$

It is obvious from equation 2.4 how such a calculation is done recursively. Meaning, in order to calculate the value of F_n, one has to calculate the Fibonacci value of the two numbers preceding it and so on. Following the tabling approach, the elements of the Fibonacci sequence already calculated are kept track of. This helps avoiding doing any re-calculations.

Algorithm 2 is equivalent to the previously mentioned mathematical relation, where X is the number the Fibonacci is being calculated for and R is the resulting Fibonacci number. The invoked method *calculated_before* on X is a boolean check on whether the X^{th} value of the Fibonacci series has been calculated before or not. *Find_Fibonacci* is a method that retrieves the resulting Fibonacci of a number that has already been calculated before and returns it in RX.

Algorithm 2 Fibonacci: Top-Down Approach With Tabling

```
Fib(X, R){
if X = 1 or X = 0 then
    R ← 1
else
    if X.calculated_before then
        Find_Fibonacci(X, RX)
        R ← RX
    else
        Fib(X − 1, R1), Fib(X − 2, R2)
        R ← R1 + R2
    end if
end if
}
```

Thus, we have the final Fibonacci of a number returned in R of the first Fibonacci call eventually.

2.2.3 Wolfram's Cellular Automaton: Rule 110

A cellular automaton is generally a group of coloured cells in a grid-like representation. Each cell, in a generation G, results from a set of rules being applied on a group of cells in the previous generation $G - 1$. Iteratively, the rules are applied for as many steps as requested.

The simplest case is having a one-dimensional cellular automaton. An even simpler class of such cellular automata, is an *Elementary Cellular Automaton*. The cells can only have two possibilities, 0 or 1, and the rules applied are only concerned with the nearest neighbour cells. One of those rules introduced by Stephen Wolfram in [10], is rule 110. Using this rule, the next colour of a cell depends only on its colour and the immediate neighbouring cells' colours. The number 110 from the rule's name represents its outcome when encoded in its binary representation; $110_{10} = 01101110_2$. Figure 2.1 illustrates what the binary number represents, where the upper group of cells are the neighbouring ones in generation G and the lower cell is the output cell in generation $G + 1$.

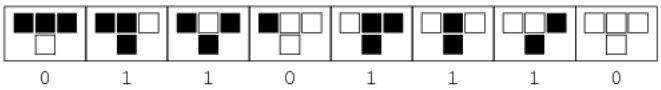

Figure 2.1: The 8 rule cases of Rule 110. Courtesy of [1]

As shown in the above figure, for each three neighbouring cells in the top row, the outcome in the next generation, shown in the bottom row, depends on their colours(black for 1 and white for 0). An example to explain how the rule works is shown next in figure 2.2, where initially only one black cell is present in the first generation, which is viewed as a column, and the rest of the generation's cells are white. The rules are applied accordingly to output a few more generations.

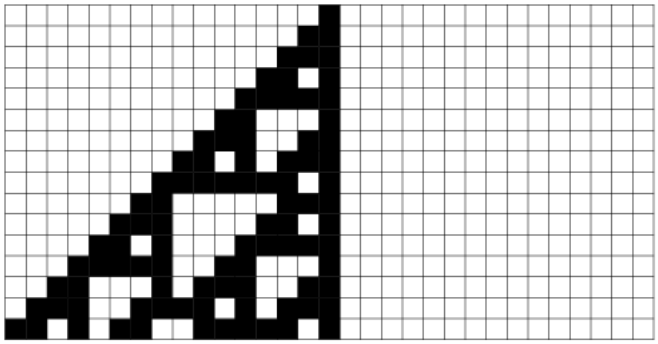

Figure 2.2: An example of how Rule 110 works. Courtesy of [1]

2.2.4 N-Queens

The N-Queens problem is concerned with placing a number of queens on a grid such that no queen could be attacking another. In other words, each queen must be placed in a cell where there is no other queen in the same row, column or diagonal. The maximum number of queens to be placed on an $n \times n$ grid/chessboard is n queens[2]. This work is interested in visualising such a case where the maximum number of queens can be placed on a board. There can be different permutations of the solution to such an N-Queen problem. For example, for eight queens, there are multiple solutions as shown in figure 2.3.

2.2.5 Sudoku

The well-known Sudoku puzzle is mainly a logic puzzle where the goal is to fill in a grid with numbers satisfying some constraints and rules. A basic "classic" Sudoku would be a 9×9 square divided into nine 3×3 boxes. One can only use numbers from 1 to 9 to fill in the grid. Initially, the Sudoku grid can have a few numbers given, provided that it satisfies the following constraints; each row, column and 3×3 box must have all nine numbers without any of them repeated. Also, filling in the rest of the board to solve the puzzle must be done fulfilling the same conditions.

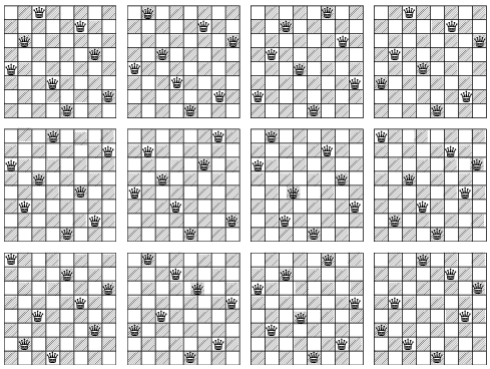

Figure 2.3: Permutations of N-Queens problem's solution with eight queens. Courtesy of [2]

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

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

Figure 2.4: A Sudoku problem on the left-hand-side and its solution on the right-hand-side. Courtesy of [3]

In figure 2.4, the puzzle begins with a few numbers in red as the input and then the constraints previously mentioned are applied on the possible numerical solutions for each cell. This eliminates the solutions that would fail to satisfy the constraints and thus leaving us with a valid correct solution to the whole puzzle.

2.3 Algorithm Visualisation

Various works have been interested in visualising and animating algorithms. According to [11], generally, an algorithm animation is done through visualising the algorithm's behaviour which can be shown through providing a step-by-step workout to the algorithm as shown in the *Sorting Out Sorting* film. In 1981, Baecker introduced this *Sorting Out Sorting* short colour sound film which explains nine different sorting algorithms through their animations. The aim of the project was to provide sufficient information about each algorithm through the visualisation so that a student watching it, with no previous knowledge about the algorithms, could be able to program some of those algorithms afterwards. According to Baecker's case study in 1991[4], *Sorting Out Sorting* has been both successful and influential among computer science students at different levels.

In 2002, the ANIMAL system[5] was interested in algorithm animation from a different aspect. The idea was to develop a system that supports multiple roles in algorithm animation. The three roles that this work was interested in are the *user*, the *visualiser* and the *developer*. This system visualised algorithms and offered different features to

each of the three roles. It enabled the *user* to alter the visualisation through various given options. As for the *visualiser*, the flexible primitives and animation effects that the base system provides can be of great help. Also, the *visualiser* is capable of loading any additions from the world wide web. The ANIMAL system gives the developer the ability to easily extend or adapt it to their preferences. This also includes adapting the language used in the graphical user interface's (GUI's) front end.

Some later works have brought the idea of visualisation to the CHR language. For example, [12] introduced *VisualCHR*; an interactive visualisation tool that visualises propagation and simplification of constraints. It is also used as a debugger for constraint solvers written in CHR which can aid the understanding of the details of the different constraints' interactions. Later, a general methodology for visualising the execution of algorithms written in CHR was proposed in [13], where the CHR program is passed to a Java application that parses the CHR code, extracting the needed information from it to output the final program showing the visualisation.

Chapter 3

Approach

In order to tackle the graphical aspect of this work, some researching was done to inspect the graphical capabilities of SWI-Prolog, since it is the chosen compiler to be used for this project.

3.1 Choosing a Graphical Approach

According to the official SWI-Prolog website[8], SWI-Prolog on its own lacks such graphical capabilities. However, a few other possible substitutions to overcome this shortage are available.

3.1.1 Using an External GUI Language

The Embedding Approach

This approach mainly presents the possibility of embedding the Prolog system in an application written in an another language, where Prolog implementations can provide potent interfaces to C and mechanisms to such embedding. Even though one can have the freedom in choosing any desired GUI platform when following this approach, it is quite difficult to debug and develop the hybrid environment in general[8].

Using Pipes

An alternative more popular approach is to exploit an external language, such as JPL/Java, that can handle GUI development and connect the interface to Prolog, i.e acting as a pipe. JPL (Java/Prolog) is a library of a collection of Java classes and C functions offering an interface between Java and Prolog through its 2-layer design; a low-level interface to the Prolog Foreign Language Interface (FLI) and a high-level Java interface for a Java programmer with no interest in such details of the Prolog FLI. Hence, the Java environment's robust class structure can easily be taken advantage of from within Prolog. Thus, this approach had been used often such as in [13] where JPL was used to run the CHR program through the Java application enabling the possibility to present a rich GUI. In addition to the JPL/Java option, other popular candidates are Tk/Tcl, Visual Basic and Delphi. However, a very restricting disadvantage to this approach is that when having intensive communication between the interface and Prolog application, performance can decrease significantly[8].

3.1.2 Using Graphical APIs

Even though it is not a commonly used approach, using an Application Programming Interface (API), written specially to meet the graphical needs of Prolog, is still a valid option. A significant example of such an API is Xwip which stands for X Window Interface for Prolog and is written in Prolog and C language. Xwip offers the ability to use Prolog as a client program for the X Windows System[14]. However, the general reason behind the unpopularity of such API's usage is that most of them are low-levelled and difficult to deal with regarding the representation of their data types in Prolog[8].

3.1.3 Using XPCE

XPCE is a platform portable tool-kit used for developing GUIs where the platforms covered are UNIX/X11 and Windows (Windows-NT/2000/XP/Vista). Its aim is to speedily develop fast and well-structured GUIs across the previously mentioned platforms. XPCE mainly offers two different ways of programming. The first simple option is to use the built-in and library classes from within one's Prolog code. On the other hand, for a larger and more well structured graphical application, one can use the normal Prolog syntax to define their own new XPCE classes to meet the application's needs.

A Closer Look into XPCE

Since XPCE's kernel is an object oriented engine, it permits the possibility to define methods in several languages. Furthermore, its built-in graphics are defined in C to enhance its speed, whereas applications and application-oriented libraries are defined as XPCE classes whose methods are defined in Prolog. Such approach for developing a GUI application allows those methods to run naturally in Prolog taking advantage of Prolog's rapid development cycle[8]. Another advantage is that when employing XPCE/Prolog, a Prolog fact-base is used for describing any relevant information about entities in one location. Such information allows the maintenance of the database and evaluation of constraints on the data.

Introducing XPCE

Assuming one's familiarity with Prolog syntax, the simplest way to introduce XPCE is through the four predicates it adds to Prolog introduced in [15].

- `new(?Ref, +Class(...Arg...))`: Creates an object `Ref` as an instance of a class `Class` using the given argument(s) `Arg`. The reference `Ref` can either be a Prolog variable or an XPCE object reference displayed as `@ref` where the name after the "`@`" can be any name. The "`?`" indicates that the argument following it is an optional input, whereas the "`+`" means that it is required.
- `send(+Ref, +Method(...Arg...))`: Similar to `new/2`, `send/2` invokes the method `Method` with arguments `Arg` on the already existing object referenced `Ref`. This predicate returns a boolean indicator. It can also take more than two arguments as to be shown in an example shortly.
- `get(+Ref, +Method(...Arg...), -Res)`: Retrieves information about an object `Ref`. It is usually used to return a 'true' result value `Res` from the method `Method`. However, when having a primitive result, the result is converted into a Prolog integer, float or atom. As for any other case, the object's reference is returned. The "`-`" as in "`-Res`" indicates that this argument is a returned value.

- `free(+Ref)`: destructs the referenced object `Ref`.

The following sample queries are examples of how one can employ the previously mentioned predicates.

```
?-  new(@p, picture), send(@p, size, size(70,70)), send(@p, display,
new(@c, circle(40)), point(30,30)), send(@p,open), get(@c, radius,R).
R = 20.

?- free(@p), free(@c).
true.
```

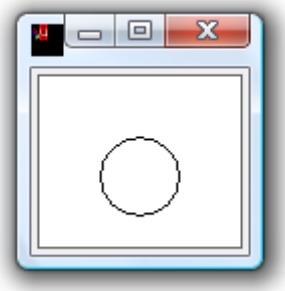

Figure 3.1: An XPCE example showing a circle drawn in a specified position of a picture

The output shown in figure 3.1 is a result of the first query. First, an object with reference `@p` and class `picture` is initialised to output a picture which is a window usually used for application graphics since it has scrollbars. The size of the picture was specified through the following `send` predicate where the method `size` was invoked on the picture `@p` with an extra argument `size(70,70)` to define its width and height.

Next, to draw a circle n the picture, another `send` predicate is invoked on the picture's reference however this time with the method `display`. The third argument of this predicate is a newly created circle whose reference is `@c` and as shown, the class `circle` takes one argument which is the circle's diameter.

Creating the picture and the circle does not mean it is shown to the user yet. Hence, the next `send` predicate invoking the method `open` on the picture, is queried. Finally, to show what a get query would output, the method `radius` is invoked on the circle to return the radius of the circle in variable `R`.

The second query destroys the picture through freeing its reference and consequently any graphical object in the picture will be destroyed as well since they are bound to it.

Choosing XPCE

One of the aims of this work was to find a stable and easy-to-employ graphical approach for the algorithms' visualisations. After exploring other possibilities, XPCE was found to be the most suitable choice. As previously discussed, when employing XPCE one can use solely Prolog to develop a graphical application. This avoids any clashes that can occur as a result of massive communication between two different languages employed together to develop the graphical application.

Also, CHR was initially ported to Prolog and ever since, Prolog has been the most supported and commonly used language with CHR. Thus, using Prolog alone ensures the stability of the code.

For the previously mentioned reasons, XPCE will be the approach used for the graphical implementation part of this work.

3.2 Adding Graphics to Algorithm Implementation

Having chosen Prolog with XPCE to be used for visualising the algorithms, the basic idea for this work is to add XPCE predicates to the CHR implementation of the chosen algorithms. In order to apply such additions, some of the already implemented CHR rules will need to be altered. Constraints, as well as the previously mentioned XPCE predicates, will be added to the head, guard and/or body of the rules to aid the visualisation of an algorithm. This will be explained in more details throughout chapter 4. However, in the following sub-sections, a brief look over each of the five algorithms' graphical approaches will be presented.

3.2.1 Fundamental Algorithms

For the following two algorithms, the graphical approach was not based on a specific method of visualisation. Each algorithm was visualised differently based on which steps of the algorithm the user would be interested to view. Also, we are concerned with illustrating each algorithm clearly to the user so that it can aid the understanding of how the CHR rules are applied.

Minimum of a Set

A simple problem such as finding the minimum in a set of numbers would not require a complicated visualisation. Most importantly, the state of a number needs to be shown. There can be three states for a number:

- The number is being compared to another. This implies that it is not yet refused nor accepted as the minimum-so-far.
- The number is marked as the current minimum-so-far after being compared to the previously marked minimum-so-far number and found to be less than it.
- The number is rejected as being the new minimum-so-far after comparing it with the current one.

The idea of visualising this algorithm is to show the numbers in a graphical shape, for example a circle, where its colour would differ according to the state it is currently in. Such choice of the visualisation is explained in sub-section 4.2.1.

Fibonacci: Top-Down Approach with Tabling

Referring back to how a top-down algorithm of calculating the Fibonacci of a number works as explained in sub-section 2.2.2, it can be useful to always keep track of the numbers and their Fibonacci number if calculated. The main idea of the graphical approach for this algorithm is to visualise a number in a rectangle divided into two parts where the left-hand-side is the number itself and the right-hand-side is the calculated Fibonacci number.

3.2.2 Grids

Regarding Wolfram's cellular automaton, the N-Queens and Sudoku problems, as shown in 2.2, 2.3 and 2.4 respectively, the common way of visualising such problems is through having a grid-like representation or board that the algorithm can run on.

Wolfram's Cellular Automaton: Rule 110

As for rule 110 from Wolfram's cellular automata, the graphical approach to be followed in this work is taken from Tom Schrijvers's slides[16]. Each generation consists of four vertical cells and the input is the user's choice. A cell can be either white or black which corresponds to 0 or 1. The constraints are run on the input cells to yield more generations. At any point, the neighbouring cells involved in a computation as well as the new cell resulting from it in the next generation, will all be marked to aid the comprehension of the algorithm's flow for the user. As for the maximum number of generations computed, it will be left for the user to specify.

N-Queens

In order to find a valid position for a queen to be placed without being attacking any other queen, different positions of the queen need to be tested. The cases of each cell of the grid can be summarised into four cases:

- **Undecided:** Meaning this cell can possibly have a queen.
- **Not a possibility:** Whenever a queen `Q` is placed in a cell, it eliminates the possibility of having another queen placed in the same row, column or diagonal of `Q`. Such positions of the eliminated cells are what this case covers.
- **Bad Queen:** A bad queen is one that was found to be in a position that does not allow other queens to be placed after it in a way that would lead to a valid solution.
- **Good Queen:** A good queen is one that is in a position such that it allows the rest of the queens to be safe, i.e not attacking each other.

The different cases of a queen's state will be visualised as a question mark, an empty cell, a red crown and a green crown respectively. The concept behind such visualisation choices will be explained later in the explanation of the N-Queens implementation in chapter 4's sub-section 4.2.4.

Sudoku

Similar to the N-queens problem, the Sudoku puzzle works based on constraints being applied on the input numbers. Initially, each cell can carry any number. This will be visualised through marking nine dots on each cell representing the possible numerical values it can take; from 1 to 9. Whenever a number is entered in a cell, the dot corresponding to that number is deleted from each cell in the same row, column and 3×3 box it is in. We are interested in showing a step-by-step workout of how the constraints are applied on the cells causing such eliminations to take place and eventually bringing us to a valid solution to the puzzle.

Chapter 4

Implementation

This chapter is concerned exclusively with the implementation part of this work. The main focus is introducing the algorithms' implementations and how the graphical implemented part is embedded in them. Each of the five algorithms' visualisations will be explained in a detailed manner with sample queries for further understanding. Note that each implementation's `.pl` file is consulted separately through the SWI-Prolog console, and using the sample queries that will be given shortly, one can run the algorithms' visualisations.

4.1 Outline of Implementation

For simplicity, a general common outline was put to be followed for each algorithm implementation. Each of the following sub-sections may or may not be present in an algorithm's code depending on the algorithmic and graphical needs of the implementation. This will be explained shortly for each of the coming sub-sections.

4.1.1 Initialising the Visualisation

Initially, there has to be a window for the visualisation to be displayed on. As stated before in 3.1.3, a `picture` in XPCE is a window with scrollbars mainly used for graphical applications. A window can also have scrollbars, however they would have to be implemented manually. Thus, for simplicity and maintaining the generalisability of the implementations, a picture was used for all implementations. For later reference, a `picture` will be referred to as a *'window frame'* in order to avoid any confusion with the meaning of an actual picture/image.

Initialising the window frame is done in every implementation through calling a certain predicate responsible for creating it. Also, such predicate can be used to initially throw any constraints that can be needed to begin the algorithm being visualised.

The following example of an initialising predicate `start/1` is taken from the N-Queens problem's implementation:

```
start(X):-
    free(@p),
    speedch(0.10),
```

```
new(@p, picture('N-Queens Solver')),
W is 100+(X*40)+((X-1)*10),
send(@p, size, size(W,W)),
send(new(D, dialog), below, @p),
send(D, append, button(speed_up, message(@prolog, speed,  -0.1))),
send(D, append, button(slow_down, message(@prolog, speed,  0.1))),
send(D, append, new(Quit, button(quit, and(message(D, destroy))))),
send(D, done_message, message(Quit, execute)),
send(D, default_button(slow_down)),
send(@p, open).
```

Generally, this predicate creates a new window frame of type `picture` with a personalised size. A `dialog` is also added at the bottom of the frame to hold the added buttons. In XPCE, a `dialog` is basically a window specialised for laying out and handling messages required by any XPCE controller that is appended to the `dialog`. For the functionality of XPCE's predicates `send` and `new`, refer to section 3.1.3.

Some Extra Features

In each window frame, there are three buttons;

- Quit: stops the visualisation and destroys the window,
- Slow Down: slows down the visualisation if one needs to keep better track of its flow in a slower pace,
- Speed Up: accelerates the visualisation. This can be used when, for example, the user wishes to reach the next step in a visualisation. It is also useful if one wants to just go through the whole visualisation quickly to arrive at the final solution.

Those buttons are added to the `dialog D` built on the window frame, through the predicate initialising the frame as discussed earlier. For example, the `speed_up` button is created through the following line of code:

```
send(D, append, button(speed_up, message(@prolog, speed,  -0.1))),
```

where a button `speed_up` is appended on dialog `D`. A `message` here is sent to `@prolog`, which is an object that allows calling a Prolog predicate, in this case `speed`, from the XPCE environment. The arity of the called predicate has to be equivalent to the number of arguments to the `message`. Applying this on the given code sample, whenever the `speed_up` button is clicked, the predicate `speed/1` is called with the argument `-0.1`.

4.1.2 Helper Predicates and/or CHR Rules

This part of an implementation holds any predicate or CHR rule that aids the visualisation but does not affect the actual algorithm. Also, to avoid repetition of commands, some general predicates are added to some implementations as helper ones. An example of the is `icon/3` which will be explained shortly. The following predicates or CHR rules in this sub-section are not necessarily present in all of the implementations.

Delays

Normally, when entering a query for SWI-Prolog to compile, it outputs the final result directly. This does not match the idea of this work which is to show a step-by-step workout to the user. Thus, the idea to accomplish that was to add delays after each point in the implementation where it is needed to show the user what has been done, i.e a checkpoint. The constraint `time1/0` was created to be added anywhere in the code that would be considered as a checkpoint. Another constraint `speedch/1` was created to carry the value of the delaying time. A `speedch` constraint, carrying a certain delaying value, is always thrown at the beginning of the visualisation through the initialising predicate such as in the example shown at the beginning of sub-section 4.1.1. The two constraints `time1` and `speedch` are employed together to fulfil the action of adding delays through the following CHR rule.

```
speedch(X) \ time1 <=> send(timer(X), delay).
```

This rule is fired whenever both constraints are in the constraint store. The constraint `time1` is thrown into the store whenever a delay is needed. This will be shown later when showing some samples from the implementation code. Moreover, when the rule is fired, a delay of the value carried by the constraint `speedch` , `X`, is performed and thus giving the viewing user a chance to follow the steps of the algorithm.

Whenever the `speed_up` or `slow_down` buttons are clicked, the predicate `speed/1` is called. This predicate throws a constraint `change/1`, passing on the delay value `X`, that the predicate holds, to the constraint.

```
speed(X):- change(X).
```

When a constraint `change` is first thrown into the store, the following rule is fired:

```
change(X), speedch(Y) <=> Y > 0.01| X1 is Y+X, speedch(X1).
```

where `X` is the delay change that will be applied to the current speed value `Y`. The idea is simply to add `X` to `Y` and replace the `speedch` constraint, currently present in the constraint store, with a new `speedch` constraint carrying the resulting delay value `X1`.

The value that `Y` is checked with in the guard, referred to as a 'check value' for further reference, differs from one visualisation to another. Its value is chosen according to tests made for the last positive value the delay can have after pressing the `speed_up` button multiple times. Speeding up the visualisation basically implies that the delay should be decreased. That is why when the `speed_up` button is pressed, the argument sent along with the predicate `speed` is a negative value which in turn throws a constraint `change` carrying that value. Through several trials, it was noticed that when having only one rule with no guard, to handle the `speedch` replacement, at some point the new resulting value is a negative number. This produced errors causing the visualisation to stop when `send(timer(X), delay)` is performed, as discussed before, since a delay cannot be of a negative value. Thus in order to avoid this, another rule is added.

```
change(X), speedch(Y) <=> Y =< 0.01, X > 0| X1 is Y+X, speedch(X1).
```

The above rule handles the case when `Y` has now reached the check value or is less than it, and the `slow_down` button is clicked. In this case, it is safe to change the delay since it will be a positive change. However, for the same case but with clicking the `speed_up` button, the case `X<0`, there are no rules for it and thus no changes will happen to the visualisation, avoiding any crashes that could happen as explained before. Note that the above two rules were taken from the N-Queens problem implementation as an example.

Icons

For the implementations that require using images, like the N-Queens and Sudoku problems, a predicate `icon/3` was added to be called wherever an icon of an image is to be shown.

```
icon(X,Y,P):- send(@p, display, new(bitmap(P)), point(X,Y)),
              send(@p, flush), time1.
```

An example of how to call such a predicate is `icon(X,Y,'example.xpm')`, where `X` and `Y` represent the position where the icon is to be added on the window frame `@p`. The image 'example.xpm' corresponds to `P` in the predicate and is converted into a newly created bitmap in the specified positions. The method `flush` is XPCE's "repaint" or "refresh" method to redraw the visualisation with the latest updates. It is noticed that after flushing the window frame, a `time1` constraint is thrown, adding a delay before the next step. This enables the user to see any graphical changes made.

Note: There are different approaches of displaying an image in XPCE. For this work, `bitmap` is used to convert the `.xpm` image directly into a graphical object. Also, since XPCE by default searches for any image in its bitmaps folder, the visualisations' images were added to that folder. This folder can be found in the SWI-Prolog installation folder. For example, if SWI-Prolog is installed in a folder "*pl*", then the path of the bitmaps folder would be "`..\pl\xpce\bitmaps`". For better organization, a folder with the name of each visualisation, was created in XPCE's bitmaps folder for each visualisation that required images.

Other Predicates

Various personalised predicates were added to some of the implementations to serve a certain need in them. Such predicates will be explained briefly through the detailed explanation of the algorithm implementations in the following section 4.2.

4.1.3 Algorithm Implementation

Each implementation of the five algorithms will be discussed in more details in the following section 4.2. However, a general idea is now given of the algorithms implementations' outline.

CHR rules

The CHR rules of the algorithms are taken from the official CHR website[17] with the exception of the Wolfram implementation. The basic idea of the Wolfram CHR implementation is taken from [16] and the rules were implemented based on it. This part of the implementation contains the CHR rules of an algorithm with a few additions to them to aid the visualisation. Such additions can be summarized into the following:

- Some constraints and predicates added to the body, guard and/or heard of a rule
- In some cases, extra arguments are added to the original constraints of the algorithm for visualisation purposes

Predicates

In some of the algorithms' implementations, such as in the N-queens and Sudoku implementations, some predicates are implemented to aid the CHR rules to solve the puzzles. Similar to the CHR rules, additions were also embedded in such predicates for visualisation purposes.

4.1.4 Implementation of Graphics

This part of the implementation has nothing to do with the basic algorithm of solving the problem but is solely concerned with the graphical aspect of the implementation. This will also be explained further for each implementation separately in the next section 4.2.

CHR Rules

The CHR rules were employed here to serve the visualisation and increase the code's efficiency. With the exception of the Sudoku problem, all other implementations have CHR rules dealing with the constraints created especially for the graphical part.

Predicates

These predicates serve the same goal as the CHR rules just mentioned. For most of the implementations, the body of the predicates of the graphical part of the code could have been added directly into the CHR rules' bodies. However, since some of the actions are repeated a lot throughout the implementation, each group of actions usually performed together were grouped into predicates. This helps minimizing the size of the implementation as well as keeping it efficient.

4.2 The Visualised Algorithms

In this section, we will go through every algorithm implementation and explain how they work.

4.2.1 Minimum of a Set

Finding the minimum of a set of numbers can easily be implemented through only one rule in CHR:

```
min(I) \ min(J) <=> J>=I | true.
```

where the numbers can be queried as `min(X)`. Simply, when having two `min/1` constraints, satisfying the condition of the guard, the one with the largest value is removed and the minimum of them is kept. This is done multiple times till all of the numbers have been compared and tested for being the minimum. However, in order to visualise it, a few constraints and predicates are added to each part of the rule for the implementation of this work. Thus, the previous rule is converted into the following rule:

```
circle(I,X), numtext(I,A,X), min(I) \
  min(J), circle(J,Y), numtext(J,B,Y) <=>
    ground(A), ground(B), ground(X), ground(Y), J>=I |
      updateC(X,yellow),updateC(Y,yellow), send(@p, flush), time1,
      updateC(X,green), updateC(Y,red), colortx(B,red ), free(J), true.
```

We will use the following sample query to explain how the algorithm works

```
:- start, min(3), min(2), min(1), min(4), min(-2), min(0.4).
```

First, `start/0` predicate is called to initialise the window frame. Also, two constraints; `valuec/3` and `valuet/3` are thrown through `start`. For this implementation, it was chosen to represent each number inside a circle through a simple visualisation where the circles are shown next to each other with a maximum of ten circles per row.

The first argument of `valuec` is the number of circles that can still be drawn in the current row. While the second and third arguments correspond to the x and y coordinates of the circle respectively. The arguments of `valuet` correspond to the same values but for the text written inside the circle instead of the circle itself. These two constraints are mainly used by the predicates `draw/7` and `writetx/8` as will be explained shortly.

Whenever a `min/1` constraint is thrown into the constraint store, the following rule is fired:

```
min(N) ==> circle(N,Y), numtext(N,X,Y).
```

which throws two more constrains, `circle/2` and `numtext/3`, into the store where `N` is the number itself, `X` acts as a reference to the text written inside the circle of reference `Y`. A text is linked to the circle it is in through having the reference of that circle as an argument in `numtext`. The first constraint thrown into the store will cause the following rule to be fired:

```
circle(N,Y) \ valuec(Mx,Fx,Fy) <=> draw(Y,Mx,Fx,Fy,Mx2,Fx2,Fy2),
                                   valuec(Mx2,Fx2,Fy2), time1.
```

which in turn calls the predicate `draw` that uses the reference `Y` of the constraint `circle` also as a reference of the circle object drawn. A circle is initially drawn in green as shown in figure 4.1. `Mx` is used as a check on the maximum circles that can be drawn in a row. If that check fails, then the next circle will be drawn in the next row and the value of `Mx` returns to its initial value. Otherwise, it is updated into `Mx2`. Fx and Fy are used to specify the location of the circle to be drawn and at the end of predicate `draw`, they are also updated to the coordinate values `Fx2` and `Fy2` of the next circle to be drawn. Since the above rule is a propagation one and the constraint `valuec` is needed in order to draw the circles, a new constraint `valuec` with the new values is thrown into the constraint store.

Similarly, the following rule works for the text of the numbers written inside each circle:

```
numtext(N,Y,Z) \ valuet(Mx,Fx,Fy) <=> writetx(N,Y,Mx,Fx,Fy,Mx2,Fx2,Fy2),
                                      valuet(Mx2,Fx2,Fy2), time1.
```

where the only extra argument in `writetx/8` over the constraint `draw` is `N`, which is the number to be written.

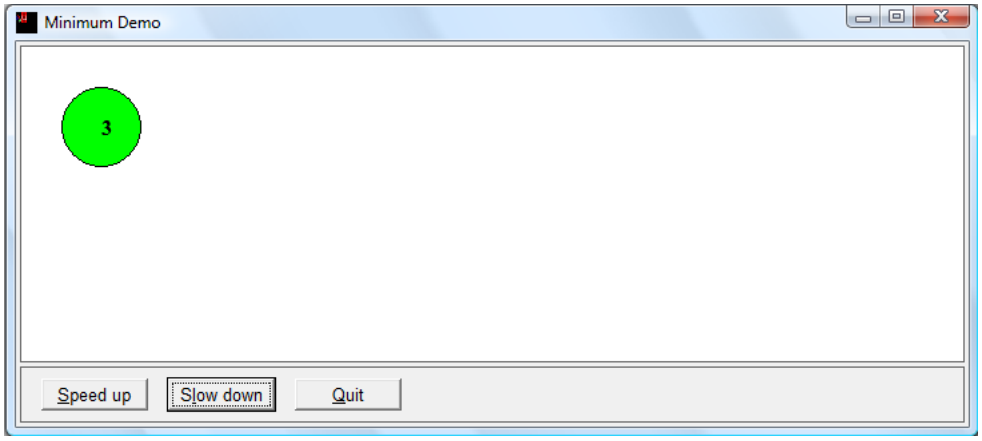

Figure 4.1: A circle initialised in green with the number written inside it.

Referring back to the main CHR rule that the algorithm actually works through, whenever there exists `min`, `circle` and `numtext` constraints for two numbers; where the references to the graphical objects, whether a circle or a text, are bound to a value, implying that they have been drawn, the rule is fired. Other than the checks done in the guard of the rule, the two circles involved in the rule, get their colours changed into yellow through the predicate `updateC/2`. This predicate, given a reference to the circle object and a colour passed as arguments, updates the colour of the referenced circle with the given colour.

Having the two circles in yellow means that currently those two numbers are being tested for which is the minimum among them. When one is found less than the other, the minimum among them has its circle's colour updated to green marking it as the minimum number found so far. Also, the other number's circle and text both get their colours updated to red, marking its failure of being a minimum candidate. Such colour representations are inspired from the different states of the traffic lights that one is used to seeing everyday.

Figures 4.2, 4.3, 4.4 and 4.5 show the different stages of the visualisation to reach the minimum of the given numbers in the previously mentioned query.

Figure 4.2: Two numbers are being compared and tested for the minimum among them.

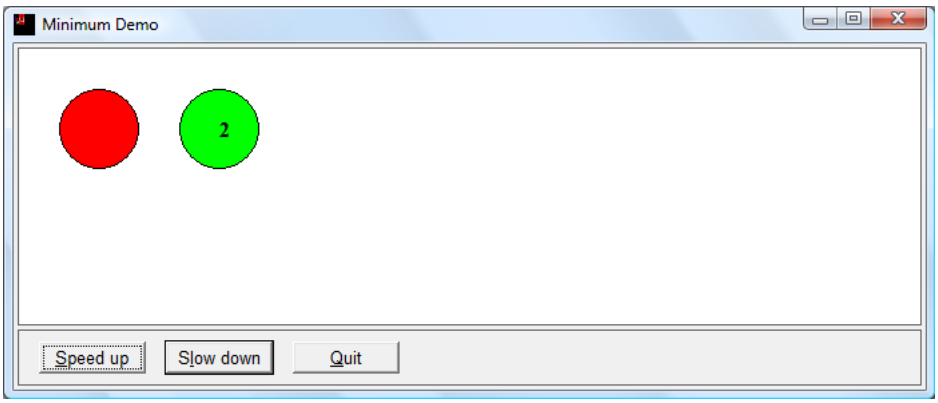

Figure 4.3: A minimum is chosen through applying constrains.

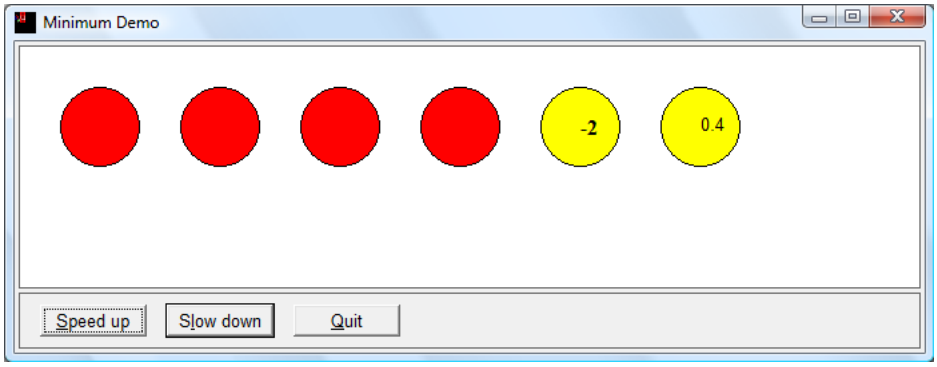

Figure 4.4: Another comparison between the current minimum and another number.

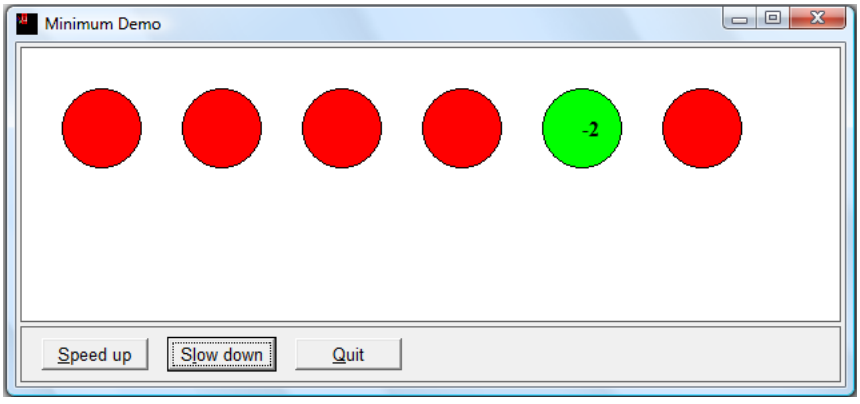

Figure 4.5: The final step of the visualisation showing the minimum of the input numbers in green.

4.2.2 Fibonacci

As previously discussed 3.2.1, the idea of visualising a number would be in a 2-part rectangle. However, the overall form of the visualisation would be in a binary tree form,

where each F_n will have a lower level with F_{n-1} on its left and F_{n-2} on its right. The algorithm mainly consists of two consecutive steps; recursively expanding the tree until the base cases are reached, then backtracking to calculate F_n.

Visualising the Nodes

The algorithm can simply be queried as follows:

```
:- start(7)
```

where it is required to calculate the seventh number in the Fibonacci sequence. The predicate `start(N)` is the initialising predicate; it creates the window frame with the buttons. The size of the frame is generic, depending on N which is the number entered by the user to run the algorithm and calculate the N^{th} term of the Fibonacci sequence. Other than initialising the window frame, the predicate `start/1` also throws in a few constraints. One of these constraints is `entry/1` where the argument it carries is N. Also, after initialising and opening the window frame, a constraint `fib/5` is thrown into the constraint store to begin the algorithm. However, before explaining further how the flow of the algorithm's implementation goes, we need to define the arguments that `fib` carries: `fib(Id,L,C,N,M)`

- N: Is a variable carrying the number where it is required to calculate the N^{th} value of the Fibonacci series.
- M: Is the calculated value of F_N.
- Id: Is a reference to `N`; acts as an identity, hence `Id`.
- L: Since the visualisation will be in a binary tree form, there will be different levels. Thus this argument corresponds to the level number `N` is on.
- C: Each node in the tree corresponds to a number X, where in order to calculate F_X, it is divided into F_{X-1} as the left child and F_{X-2} as the right child. `C`'s value corresponds to the child position with respect to the parent node; 1 for left, 2 for right and 0 in the case of the root node.

Whenever a `fib` constraint is thrown into the constraint store, there are two cases for the number N that were handled separately to preserve the simpleness of the implementation and avoid any complications:

1. **It is a number other than 0 or 1:**
   The following rule is matched and fired whenever a new constraint `fib` is thrown into the constraint store with `M` still being a variable and `N` is a number greater than 1 to ensure that the case is matched:

   ```
   fib(Id,L,C,N,M) ==> var(Id), var(M), N>=2|
                       drawparent(Id,L,C,N,_,_), N1 is N-1, N2 is N-2,
                       L1 is L+1, fib(_,L1,1,N1,M1), fib(_,L1,2,N2,M2),
                       M is M1+M2.
   ```

After meeting the conditions of the guard, `drawparent/6` constraint, which will be explained shortly, is thrown into the constraint store as well as two new `fib/` constraints corresponding to the two children. The variables `M1` and `M2` correspond to F_{N-1} and F_{N-2} respectively. Hence having `M` evaluated eventually as their sum.

As for the constraint `drawparent`, it is used solely for graphical purposes. It carries six arguments where the first four are the same as the first four arguments of `fib/` and the last two arguments, X and Y, correspond to the x and y position of the rectangle to be drawn. There are also two cases to be covered regarding drawing the node.

(a) **It is the root node:**
The rule fired in such a case is

```
drawparent(Id,L,_,Num,X,Y), entry(N) ==> L=:=1, var(X)|
      totalX(N,Z), X is ((Z/2)-20), Y=20, drawfirst(Id,Num,X,Y).
```

The guard checks that $L = 1$, i.e its the root in level 1. It also checks that X is still a variable, to avoid redrawing the node, since X is set to a value through the body of the rule. After, X and Y are calculated, `drawfirst/4` predicate is called to draw two squares next to each other, forming a rectangle. The left-hand-side square is in red and has the number Num written on it while the right-hand-side initially is green and has a question mark indicating that F_{Num} is not calculated yet such as in figure 4.6.

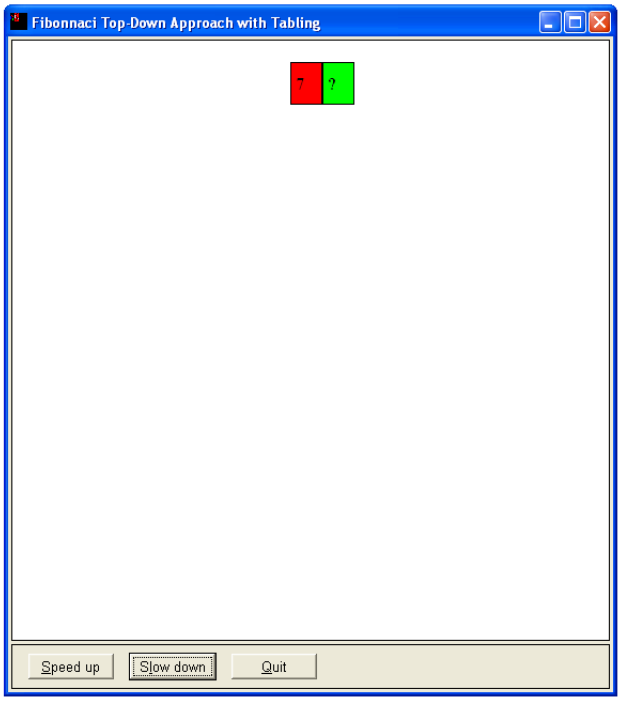

Figure 4.6: An example of calculating the Fibonacci of 7. Root node

.

(b) **It is any other node:**
For this case, the rule fired will be

```
drawparent(Id,L,C,N,X,Y), drawparent(_,L1,_,_,X1,Y1)
```

```
==> L=\=1, var(X), var(Y), ground(X1), ground(Y1),
    L2 is L-1, L1=:=L2| drawchild(Id,L,C,N,X1,X,Y).
```

Since the root node is not being dealt with any more, this node has to be a child of a parent. What this rule does is search for the parent of the node and draw the node based on the parents position. Here comes the use of the constraint `drawparent` where it can be used as a reference to any node drawn, carrying its level, numeric value, position coordinates and which child it is, at all times. The guard's checks are mainly concerned with making sure that this node has not been drawn before, is not a root node and that it actually has a parent. Satisfying those conditions leads directly to the body with the predicate `drawchild/7`. This predicate uses the level of the child `L` along with the x-position of the parent `X1` and which child `C` the node is to the parent, to calculate the positions `X` and `Y` of the node and draw it. It is noticed that `X` and `Y` are not variables any more after drawing the node, thus this rule can never be fired again for the same node. Hence, the guard check in the guard.

2. **It is 0 or 1:**
   As previously mentioned, the base cases are 0 and 1. Meaning, they can either be a root node having the rest of the tree empty, or they can be a leaf node. The reason behind not having a third case is because, for this work, the tabling approach is used as mentioned in section 4.2.2. Similar to the previous case, there is a specialised rule for having a node of value 1 or 0.

   ```
   fib(_,L,C,N,M) <=> N<2 | M=1, drawleaf(L,C,N,1).
   ```

   A `fib` constraint with its $N = 0$ or $N = 1$, is simplified into the constraint `drawleaf/4`, similar to `drawparent` with the last argument corresponding to $F_N = 1$. Furthermore, there are two cases for drawing such a node;

   (a) **It is the root node:**

   ```
   entry(N) \ drawleaf(L,_,N,Val) <=> L=:=1 |
       totalX(N,Z), X is ((Z/2)-20), Y=20, firstleaf(N,Val,X,Y).
   ```

   The guard checks that this is the first level, i.e its the root node. The position is calculated for a root node and the predicate `firstleaf/4` is called to draw the root similar to what `drawfirst` does. The only difference is that in the case of N being a 0 or 1, the value of F_N, `Val`, is already known, thus it is written in the visualisation as a 1 directly instead of a question mark.

   (b) **It is a leaf:**
   In the case of having a parent to this node, there is a rule that functions similar to the rule of the case when the number is not a 0 or 1.

   ```
   drawparent(_,L1,_,_,X1,Y1) \ drawleaf(L,C,N,Val) <=>
          L=\=1, ground(X1), ground(Y1), L2 is L-1, L1=:=L2 |
            leaf(L,C,X1,N,Val).
   ```

   Here, the same steps are taken to visualise such node. It is checked that it is not a root node and that it has an already drawn parent. Instead of calling `drawchild` predicate, the predicate for this special case is `leaf/7` where the value of F_N is already written directly as previously mentioned since it is known. Figure 4.7 shows the progress of the visualisation until it reaches a leaf.

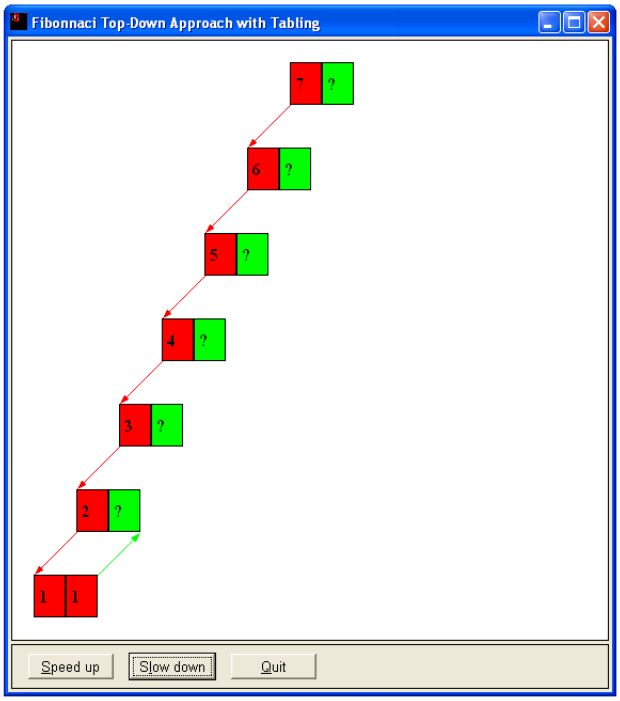

Figure 4.7: An example of calculating the Fibonacci of 7. A leaf is reached
.

Updating a Node

Since a leaf already has its value calculated, backtracking takes place to calculate Fibonacci of the second child, which is a 0 and also has a ready value equals to 1. Now for the number 2 on level 6, the M of its `fib` constraint is now bound to a value equivalent to the sum of F_1 and F_0. This will lead to firing the rule responsible for replacing the question mark in the box of a number with its now-available value.

```
fib(Id,L,C,N,M), drawparent(Id,L,C,N,X,Y), drawparent(_,L1,_,_,X1,Y1)
         ==> ground(M), ground(X), ground(Y), ground(X1), ground(Y1),
             L2 is L-1, L1=:=L2| update(X,Y,X1,M).
```

This rule is fired whenever there is a `fib` constraint having M bound to a value as well as having two `drawparent` constraints. The first one corresponds to the node of the number `N` having its F_N ready and the other corresponding to the parent node, provided that their `X` and `Y` coordinates are known. The `update/4` predicate simply updates the value of F_N and draws a green arrow from its node to the parent node indicating that this value is ready to be used by the parent4.8.

Tabling

Since F_{N-1}, i.e the left child, of a number N is always calculated first, by the time backtracking starts taking place to calculate F_{N-2}, i.e the right child, it would have already been calculated along the way through the left child's tree. Thus, in order to avoid repetitive calculations, the tabling approach is used. The following rule is added to serve such approach.

```
fib(Id1,L,C,N,M1), fib(_,_,_,N,M2), drawparent(_,L1,_,_,X1,Y1)
      ==> L2 is L-1, L1=:=L2, ground(X1), ground(Y1), ground(M2),
          var(M1)| drawchild(Id1,L,C,N,X1,X,Y),
                   update(X,Y,X1,M2), M1=M2.
```

The main idea behind this rule is that whenever there is a constraint `fib` thrown into the constraint store with $N1$ and there is another `fib` constraint in the store with $N2$ where $N1 = N2$, the node is drawn and have its F_N updated directly. The value of such update is the same as the second constraint's F_N, thus is done without any calculations as shown through figure 4.8. Finally in figure 4.9, the algorithm stops where the root node N has its F_N calculated.

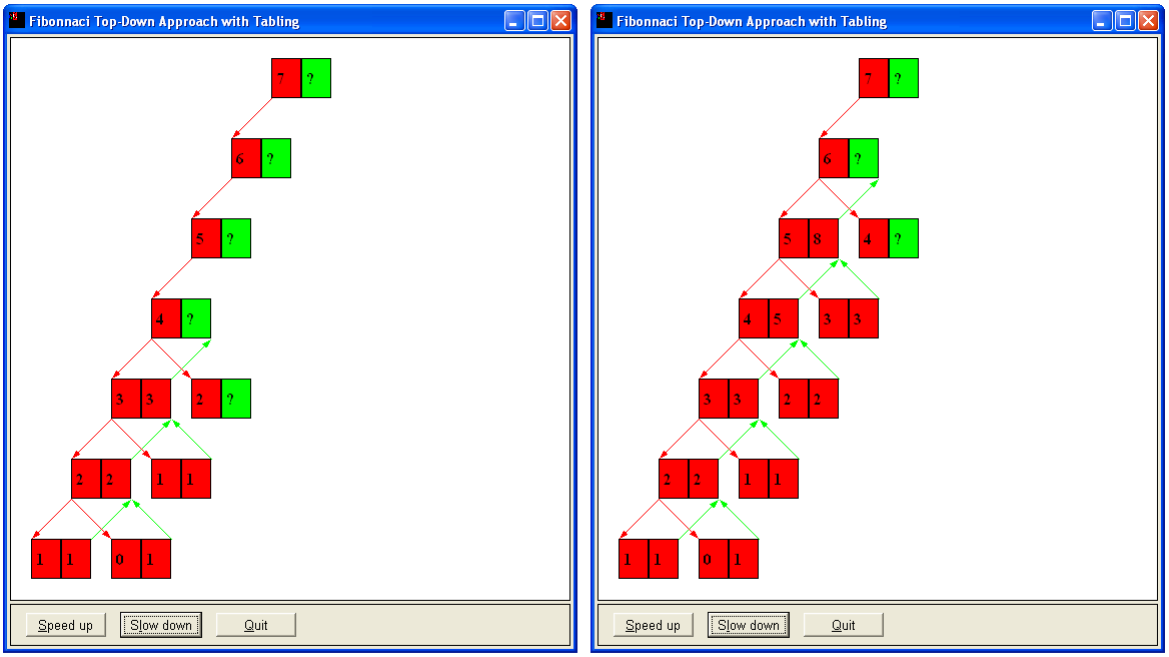

Figure 4.8: An example of calculating the Fibonacci of 7. Backtracking and Tabling.

.

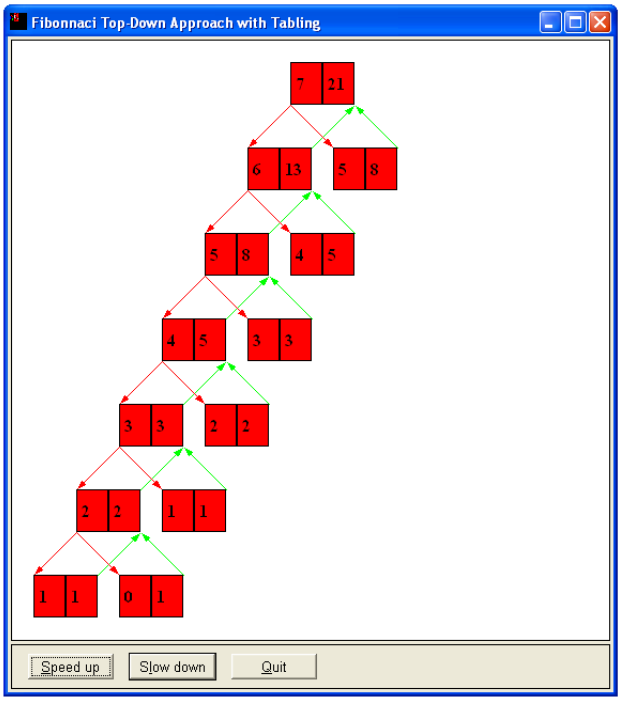

Figure 4.9: An example of calculating the Fibonacci of 7. Final solution.

.

4.2.3 Wolfram

The idea behind Wolfram's cellular automaton using rule 110 is basically applying constraints on the cells of a certain generation to output cells in the next generation. For this work, each generation was restricted to have four cells. However, there is no limit to the number of generations. We will explain how the algorithm along with its visualisation would work through showing what happens when querying the following:

```
:- wolfram(6), white(d,c,a,0), black(c,b,d,0), white(b,a,c,0),
   white(a,d,b,0).
```

Initially the predicate `wolfram/1` is queried to initialise the window frame where the visualisation will be shown. The argument given to this predicate corresponds to the number of generations G the user wants the algorithm to reach, where in this query $G = 6$. A constraint `generation/1` is thrown through the predicate `wolfram`, where the argument it carries is G. This constraint is always left unaltered in the constraint store to act as a reference at all times of G. Another constraint thrown is `yp/1`, which stands for the y-position of the next cell to be drawn. This constraint was mainly created to aid the visualisation of the initial cells. Initially it is thrown with the value of the argument equals 30. So far, no rule is fired. However, once the constraints `white/4` is thrown into the store through the query, the following rule fires:

```
white(_,_,_,0) ==> white(_,_,_,0,_).
```

A constraint `white/5` is thrown into the constraint store. The reason `white/4` propagates to `white/5` is to first draw the initial generation. The need of an extra argument will be explained shortly. Since there is a `yp(30)` constraint in the constraint store, the first rule of the following two rules is fired:

```
white(_,_,_,0,C), yp(Y) <=> Y<135, var(C)|
      send(@p, display, new(box(30,30)), point(20,Y)), send(@p, flush),
      C=1, Y1 is Y+35, yp(Y1).
```

The rule above are implemented especially to handle the case of drawing the first three cells of the first generation. Each cell is represented as a 30×30 box, with a gap of 5 pixels left between each box. Thus, the y-position of the fourth most bottom cell would be equal to 135, hence the checking condition in the guard. Also, another check is on `C` to make sure it is a variable and not bound to any value. This check ensures that the rule does not keep firing in an endless loop since in the body of the rule, `C` gets bound to a random value, in this case chosen to be 1. Furthermore, since this is a simplification rule, there has to be a `yp` constraint thrown into the store with the new y-position value `Y1` to be used when drawing the next cell and so on.

```
white(_,_,_,0,C), yp(135) <=> var(C)|
      send(@p, display, new(box(30,30)), point(20,135)), send(@p, flush),
      C=1, done.
```

The only difference between the above rule and the previous one is that the above one deal with the case of the fourth, and last, cell of the initial generation. It is noticed that at the end of this rule, `yp` constraint is not thrown into the constraint store since it will be of no use again. Instead, a new constraint `done/0` is thrown. This constraint is to mark that the initial cells have been drawn and the algorithm should start applying constraints on the input to give us a solution of the problem. This will be shown shortly through the coming CHR rules. Note that similar to the three previous rules mentioned for `white/4` and `white/5` constraints, there are rules for the constraints `black/4` and `black/5`.

For the actual implementation of the algorithm, as mentioned before, the basic idea behind it was taken from [16] then all permutations of the cells with respect to their colours, was implemented. The following CHR rule is an example of one of the permutations.

```
done, white(C,_,_,G), black(B,A,C,G), white(A,_,_,G), generation(J)
   ==> G < J | freeing, marking(C,G), time1, marking(B,G), time1,
               marking(A,G), time1, NG is G + 1, drawanswer(B,black,NG),
               time1, black(B,A,C,NG).
```

To clarify how the rule works, the explanation is divided into the three parts of the rule:

1. **Heads:**
   - `done`: As previously explained, this constraint marks that the initial input generation has been drawn. Thus, this rule, and the rest of its permutations, would not work unless this constraint is found in the constraint store.
   - `white/4`: This constraint corresponds to a 'white' cell, where the four arguments are the reference of the cell itself, the cell above it, the cell below it and the generation number that the cell is in respectively. The first three arguments can either be `a`, `b`, `c` or `d` where `a` is the bottom cell and they go upwards in ascending order to reach `d` being the top cell.
   - `black/4`: Similar to `white/4`, this constraint carries the same arguments. The only difference is that it corresponds to a 'black' cell.

In the head of the above rule, it is made sure that the cells involved are all in the same generation through having the same variable `G` as the last argument.

2. **Guard:** The guard of the rule mainly checks that the generation of the cells in the head has not exceeded the maximum number of generations entered by the user initially through checking `G<J`.

   - `freeing/0`: For the least use of graphical objects' references, this predicate is employed to 'free' the references of some graphical objects, drawn through `marking/2` and `drawanswer/2` predicates, with every rule firing. This allows the re-use of such references and thus decreasing the amount of references used. This will be explained further through explaining the following two predicates.
   - `marking`: This predicate is used to 'mark' and keep track of the cells currently being visited by a rule. It draws a small circle in the middle of the cell. Since the rules run consequently, such marking s only needed for one rule at a time; the circles can be removed once a new rule is fired. Hence, the possibility of freeing them as previously mentioned. The colour of a circle would either be one of two cases:
     (a) **Yellow:** marking the cells that caused the rules to fire.
     (b) **Red:** marking the output cell which will be explained shortly through the explanation of the body of the rule.

3. **Body:** After meeting the guard conditions, the only thing left is to draw the resulting cell, according to rule 110. To ensure the visualisation goes in the correct order, the predicate to draw the cell is called first before throwing the actual constraint of the new cell. This is done to avoid having the newly generated constraint being fired with other rules before its corresponding cell is even visualised.

   - `drawanswer`: Given three arguments, P, C and G for example, this predicate simply draws the output cell; where P the letter of the cell to help identify the y-position, C is the colour of the cell and G is the generation it is in which in turn helps identifying the x-position. Also, it marks the cell with a red circle to point it out to the user, making it easier follow the visualisation. Similar to `marking`, the circle's reference can also be freed as it will not be needed once another rule is fired.

The rules of different permutations keep firing depending on the available constraints in the store until the maximum number of generations is reached. The following figures show different intervals of the visualisation's flow.

At the beginning, the four cells of the initial generation are drawn and the first three cells of them are marked. Then, the output cell of the fired rule is drawn and marked. New constraints resulting from rules' firing cause more rules to fire and thus creating more generations. Finally, the number of generations specified by the user is reached and the visualisation stops.

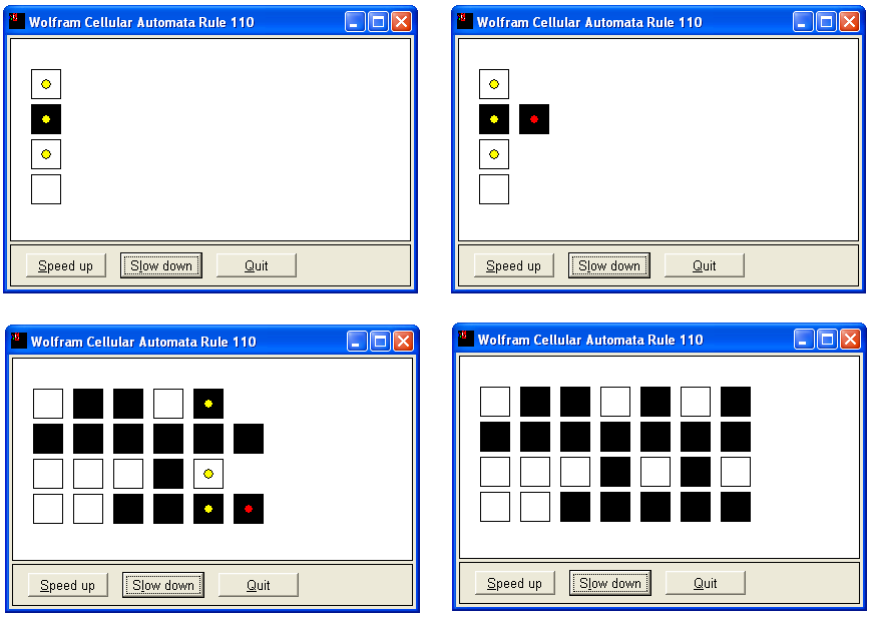

Figure 4.10: From left to right and top to bottom, an example of Wolfram's Cellular Automaton: Rule 110

.

4.2.4 N-Queens

The idea behind solving the N-Queens problem is mainly making choices and eliminations as a result of applying specific constraints on the queen positions that would ensure that none is attacking the other. In this work, showing the user how such eliminations are done through the visualisation that would lead eventually to a solution is a main concern.

As an example, we will explain how the algorithm works for solving a 4-Queens problem. The following query is given to SWI-Prolog:

```
:- solve(4,Qs)
```

where `solve/2` constraint takes the number of queens as an input and outputs their positions in the list Qs. It is simplified through the following CHR rule

```
solve(N,Qs) <=> start(N), makedomains(N,Qs), N1=N, queennum(N1),
                queens(Qs), copylist(Qs,Q2), copylist(Qs,Q3),
                queen1(Q3), queen(Q2), enum(Qs).
```

1. `start/1`: a predicate that initialises the window frame that the visualisation is going to be shown on.

2. `makedomains/2`: this predicate mainly creates the domains of each queen through the following steps:.

   ```
   makedomains(N,Qs) :- length(Qs,N), upto(N,D), domain(1,N,Qs,D).
   ```

First, `length/2` specifies the length of the list Qs to be N. Then, `upto/2` takes N as an input and outputs D which is a list of the domain [N...1]

```
upto(0,[]).
upto(N,[N|L]) :- N>0, N1 is N-1, upto(N1,L).
```

Afterwards, `domain/4` is called. This predicate which throws two constraints, `in/2` and `idof/2`, for each queen into the constraint store. The first constraint is used for algorithmic purposes while the second one is implemented for visualisation purposes.

```
domain(_,_,[],_).
domain(N1,N,[Q|Qs],D) :- N1=< N, Q in D, idof(N1,Q), N2 is N1+1,
                         domain(N2,N,Qs,D).
```

- `in/2` throws constraints that look like this: `Q in D` where Q is the reference to a queen and D is its domain. For example, there would be $Q1$ in [1,2,3,4].
- `idof` constrains carry two arguments; the number of a queen and its reference. For example, $Q1$ refers to the first queen, thus there would be an `idof(1,Q1)` constraint. This was implemented for simplicity in the visualisation's implementation part; where the first argument is a reference to which column of the grid the queen is supposed to be on. For example, queen 1 is on column 1, queen 2 is on column 2 and so on.

3. `queennum/1`: a constraint thrown into the store and carries the number of total queens. It is used as a reference at all times to that number.

4. `copylist/2`: a predicate that, given a list Qs, makes a clone of the list into $Q2$. Since the visualisation and the algorithm implementations run in parallel, at some points it was needed to have the same list doing different actions for the different parts of the implementation. Thus, in order to preserve the values of such a list for each need of the implementation, this predicate serves such issue.

5. `queens/1`, `queen1/1` and `queen/1`: are all constraints carrying a list of queen variables. `queens` is used by the algorithm implementation and is the only one carrying the original list to be returned at the end of the algorithm. As for `queen1` and `queen`, they carry clones of that list and are used for different visualisation purposes.

   (a) Upon throwing the constraint `queens/` into the store, the following rule is fired:

   ```
   queens([Q|Qs]) <=> safe(Q,Qs,1), queens(Qs).
   ```

   where the constraint `safe/3` simplifies into:

   ```
   safe(X,[Y|Qs],N) <=> noattack(X,Y,N), N1 is N+1, safe(X,Qs,N1).
   ```

   This eventually results in throwing `noattack/3` constraints into the store for each queen where X and Y are references to queens and N corresponds to the column distance between them. This implies that queen Y must not be in an 'attacking' position from queen X. This constraint will be responsible for narrowing down the domains of the queens as will be explained later on.

   (b) Once `queen` is thrown into the constraint store, the following rule would have all of its head's constraints in the store and thus is fired:

```
idof(X,Q), Q in L , queennum(N) \ queen([Q|Qs]) <=> var(Q)|
    N1 = N, drawme(X,L,N1), send(@p, flush), time1, queen(Qs).
```

For each queen Q_n, the predicate `drawme/3` is called with the column number X of the queen, its domain list L and the total number of queens $N1$ indicating also the maximum number of rows and columns the grid can have. This predicate draws or updates the grid's cells according to the domain of the given queen. Generally it has two cases for the image displayed on the cell:

- A question mark: indicating that the number corresponding to this cell is in the domain, where generally 1 is the highest cell and N is the lowest. The reason a question mark is chosen for this case is because it gives the user viewing the visualisation a sense that this cell is not known yet whether it will work or not as a position for the queen. However, it is a possible solution at this point of the visualisation.
- An empty box: meaning that this cell is not a number in the domain list. The empty box helps the user to eliminate the thought that this cell is a possible solution.

As a result of firing this rule, multiple times for each queen, the visualisation reaches the following step shown in figure 4.11, where all of the queens still have their domains $[1...N]$

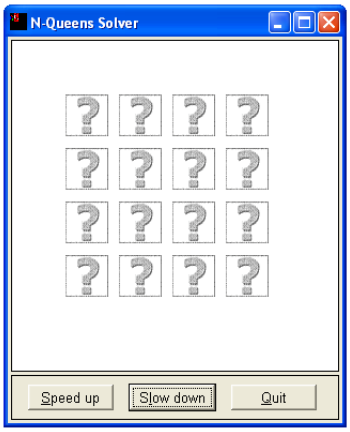

Figure 4.11: Initial domains of a 4-queen problem.

6. `enum/1`: This constraint is the one responsible for beginning the eliminations and thus leading to reach a solution to the problem. Once `enum` is found in the store, the following rule, which will be referenced as the 'enum-rule', fires:

```
queennum(M), idof(N,X), queen1(Qs) \
    enum([X|R]), X in L <=> enum_val(M,N,Qs,X,L), enum(R).
```

where given a certain queen reference X, its number N and domain L along with other arguments, `enum_val/5` predicate begins the eliminations, both algorithmically and visualisation-wise.

```
enum_val(M,N,Qs,X,[V|L]) :-
           (N1 is M, drawqueen(N,V,'nqueens/green2.xpm'),
```

```
        delrest(N,V,N1), send(@p, flush), time1, X=V) ;
       (drawqueen(N,V,'nqueens/red2.xpm'), time1,
        enum_val(M,N,Qs,X,L)).
```

First, the case of equating X by V is tried out. This basically means that the current trial will be giving queen X a value of V which is the first number in its domain. For example, initially the queen that will be matched using the above CHR rule and predicate, is $Q1$ with the value 4. As a result, an image of a crown inside a green circle will be shown in position 4 of the first column to mark that this queen is in a valid position at this point. This is done through the predicate `drawqueen/3`. Then, the predicate `delrest/3` is used to delete, i.e put empty boxes, the rest of the cells of $Q1$'s column. The assignment $Q1 = 4$ will lead to firing this next rule:

```
queennum(N1), idof(M,Y) \ noattack(X,Y,N), Y in L
        <=> number(X), X1 is X-N, X2 is X+N, delete(L,X,L1),
            delete(L1,X1,L2), delete(L2,X2,L0), L\==L0 |
                drawme(M,L0,N1), send(@p, flush), time1, Y in L0.
```

where `number/1` is a boolean check on its argument that it is a number and `delete(List, Num, List1)` predicate deletes the number `Num` from list `List` and outputs a new list `List1`.

Since $Q1$ is the only queen assigned to a value so far, the `noattack` constraints with $Q1$ as the first argument will be matched and fired consequently with this rule. For example, the head will be matched as the following:

```
queennum(4), idof(2,Q2) \ noattack(4,Q2,1), Q2 in [4,3,2,1]
```

$X1$ and $X2$ are calculated and, in this case, are equal to 3 and 5 respectively. The next step is to eliminate those values, if they exist, from the domain L and have a new domain $L0$ for queen Y. The visualisation is updated here through giving `drawme` $Q2$'s number M and its current domain. This rule is fired for the rest of `noattack` constraints having $Q1$ as an argument as well and the following step in the visualisation is reached:

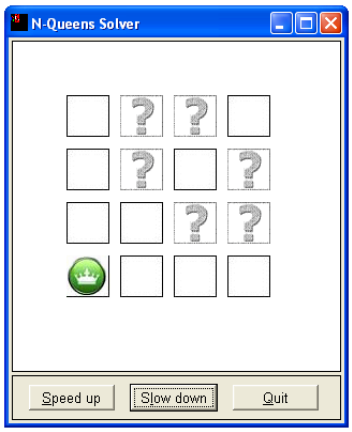

Figure 4.12: The first queen's position is set and eliminations are done accordingly

.

At this point, the algorithm backtracks to the 'enum-rule' and evaluates `enum` for the rest of the queens list Qs. This leads to repeating the same process of the `enum_val` predicate for the second queen as shown in figure 4.13.

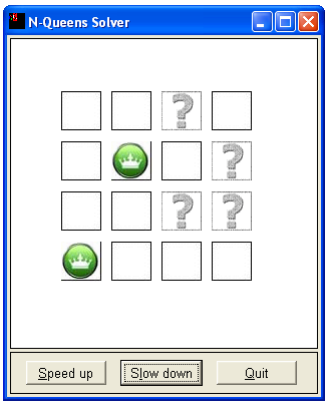

Figure 4.13: The second queen's position is set.

.

However, when firing and applying the `no_attack` rule, the third queen's domain becomes empty. Meaning there is a constraint `Q3 in []` in the constraint store. Such case matches the head of this rule:

```
queennum(M), idof(N,X) \ X in [] <=> drawme(N,[],M),
                                     send(@p, flush), time1, fail.
```

Thus, the column of $Q3$ is updated to have all of its cells as empty boxes indicating the failure to have a possible valid queen. Such failure causes $Q2$'s `enum_val/5` predicate evaluation to fail; meaning the choice of setting $Q2$ with the first value of its domain was unsuccessful. This brings us to the second part of the predicate which acts as an alternative in case the first part fails:

```
(drawqueen(N,V,'nqueens/red2.xpm'), time1, enum_val(M,N,Qs,X,L))
```

Instead of a crown in a green circle marking the position of the second queen, it is first updated to be a in a red circle indicating its failure as shown in figure 4.14.

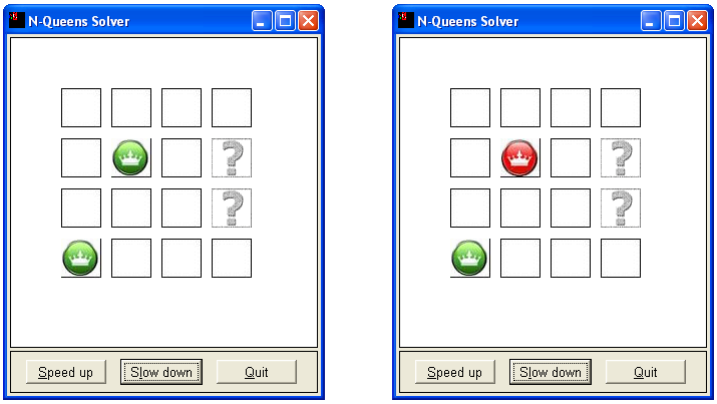

Figure 4.14: The domain of the third queen becomes empty. The second queen fails.

Then, the algorithm backtracks to the alternative choice which is choosing the next value in the domain of $Q2$ and re-run the same process on it. This is shown in figure 4.15 and is done by calling the predicate `enum_val` with the list argument passed as the rest of the domain list of $Q2$.

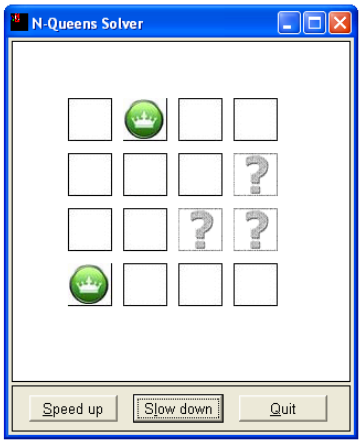

Figure 4.15: The next option of the second queen's position is tested.

With the new position of $Q2$, a resulting constraint of $Q3$'s value, after firing the `no_attack` rule, is `Q3 in [3]`. This leads to firing the following rule;

```
X in [X1], idof(N,X) <=> drawqueen(N,X1,'nqueens/green2.xpm'),
                         X=X1, send(@p, flush), time1.
```

which handles the case of having only one value in the domain of a queen, i.e only possible position. Therefore, that queen is set to a value and is drawn accordingly as visualised in figure 4.16.

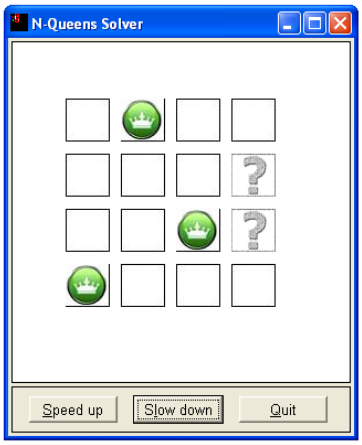

Figure 4.16: The third queen is set to a certain position since it is the only valid one.

However, when having the first three queens set to the values they currently are set to, and after firing more `no_attack` rules based on the updates reached by

the algorithm, it is found that now $Q4$'s domain is empty as shown in figure 4.17. Thus, this leads to the failure of all three positions of the queens.

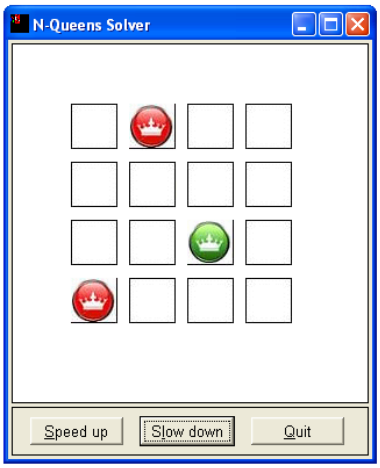

Figure 4.17: The fourth queen has an empty domain leading to the failure of the previous three queens.

This backtracks to the very first queen and the next value of its domain is tried out. The same steps of the algorithm are repeated all over again for the newly set value of Q1 until a solution is reached as shown in figure 4.18.

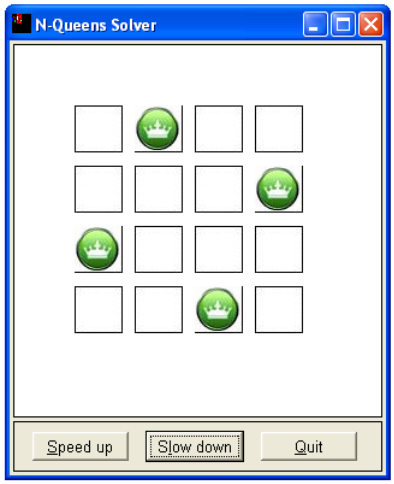

Figure 4.18: The first solution found to the 4-Queen problem.

For an alternative solution (if exists), using SWI-Prolog's command ";" will enforce the algorithm to run again to reach a different solution. For this example, it will result into the solution shown in 4.19

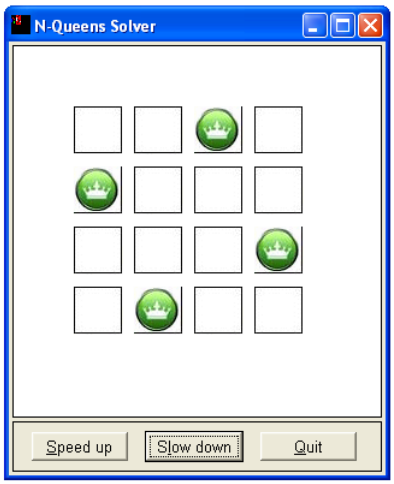

Figure 4.19: An alternative solution to the 4-Queen problem

4.2.5 Sudoko

The algorithm behind solving the Sudoko problem, taken from [17], is quite similar to the N-Queen's algorithm. Both of them are based on applying constraints that causes some eliminations. In case of the N-queens, eliminations are done on each queen's domain. However, for the Sudoko algorithm, the eliminations are done on the domain of each cell as will be discussed shortly. The basic idea behind the algorithm is having domain lists for each cell with the possible numerical values, from [1...9], that it can take depending on the values or domains of other cells.

Unlike what one is used to when solving a Sudoku problem, the implementation on the CHR website [17] solves an empty Sudoku. Meaning, the algorithm manually sets the values of a few cells initially and then the constraints are applied on the rest of the cells accordingly. To query the algorithm, the following query is used:

```
:- solve.
```

where the predicate `solve/0` has a body as follows

```
solve :- start, init_board, init_data,trace, try.
```

1. `start/0`: a predicate that initialises the window frame and calls another predicate `drawcells/2` initially given the two arguments 1 and 1. The grid is drawn through `drawcells`, in a matter that would enable the user to see the rows, columns and boxes of the Sudoku board easily as presented in figure 4.20.

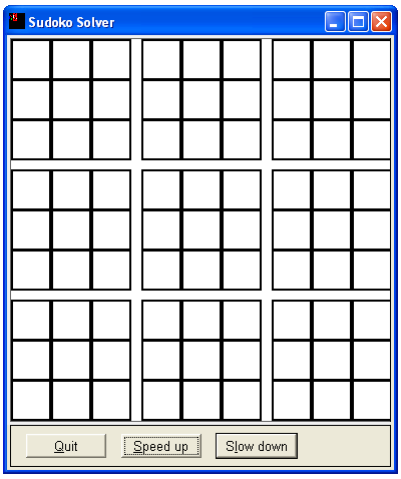

Figure 4.20: An empty Sudoko board drawn initially.

2. `init_board/0`: this predicate calls a series of predicates sequentially to eventually throw `cell/6` constraints for each cell on the board and call the predicate `updatecell1/7` for each one as well.

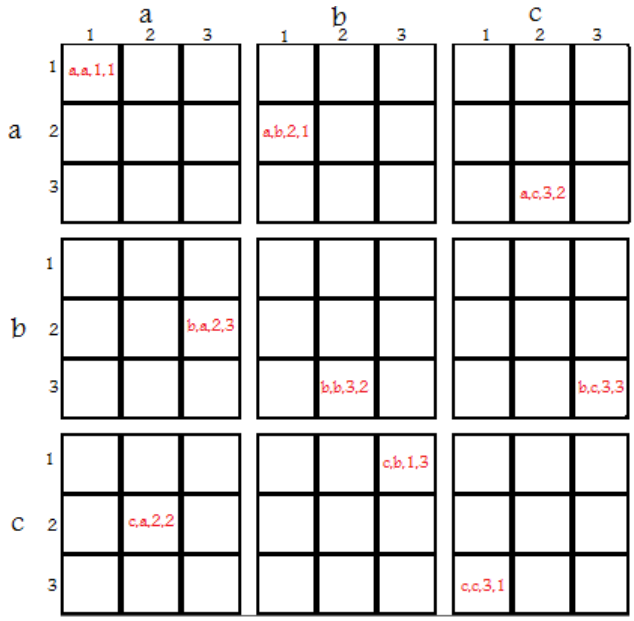

Figure 4.21: An illustration of how a cell can be referenced.

A `cell/6` constraint would carry the following arguments respectively, where the first four define the position of the cell:

- A: the letter corresponding to the vertical box the cell is in.
- B: the letter corresponding to the horizontal box the cell is in.
- C: the number corresponding to the vertical position of the cell in the box (A, B).

- D: the letter corresponding to the horizontal position of the cell in the box (A, B).
- N: the number of items in the list L; i.e its length.
- L: a list containing the possible values that this cell can take. It acts as the domain of that cell.

The figure 4.21 illustrates how the lettering and numbering is mapped on the board. Also, some of the cells have their position's reference written inside them in red as an example on how one can locate a cell.

When the constraint `cell/6` is thrown for each cell, having L initially set to $[1, 2, ...8, 9]$, the predicate `updatecell1` is also called for each cell with the same arguments as `cell/6` in addition to an extra one for the colour. The need of having this seventh argument in `updatecell1` will be explained shortly.

```
updatecell1(A,B,C,D,N,L,Col):- clearcell(A,B,C,D),
                               updatecell(A,B,C,D,N,L,Col).
```

The predicate `updatecell` calls two predicates:

- `clearcell/4`: given the location of the cell through its four arguments, it clears the cell. This is done through placing an empty box at the cell's position. Such predicate is useful in later stages where a cell could already be occupied with certain graphical objects and it is required to update the cell. Thus, one can just clear the cell and re-fill it with the updates.
- `updatecell/7`: basically updates the cell, whose position can be interpreted through the first four arguments, with its current domain L. As mentioned before, N is the length of the list L. Depending on the value of N, one of the two definitions of the predicate `updatecell` is chosen.
  (a) `updatecell(A,B,C,D,N,[H|T],_)`: deals with the case $N > 1$, implying that the domain list of this cell contains more than one value; there are different numerical possibilities for the cell. For this case, the values are represented as black dots where their position indicates the value of the number. Each dot is originally an image that is just placed in a position depending on the value of the number. Figure 4.22 shows how the numbers would be mapped as dots.

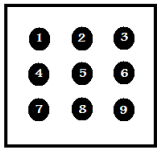

Figure 4.22: A maximized cell showing how the mapping of numbers to dots.

Note: the numbers written in white inside each black dot are only shown in the above figure for explanation purposes and not the case in the implementation's visualisation. Instead, only the black dots are shown.

(b) `updatecell(R,C,NR,NC,1,[H],Col)`: this definition of the predicate only works for the case $N = 1$, meaning that the cell currently has only one possible value it can take. Thus, instead of showing a dot, the number itself H is written as a text inside the cell in the colour Col.

The resulting visualisation after calling the predicate `updatecell1` for all cells is shown in figure 4.23:

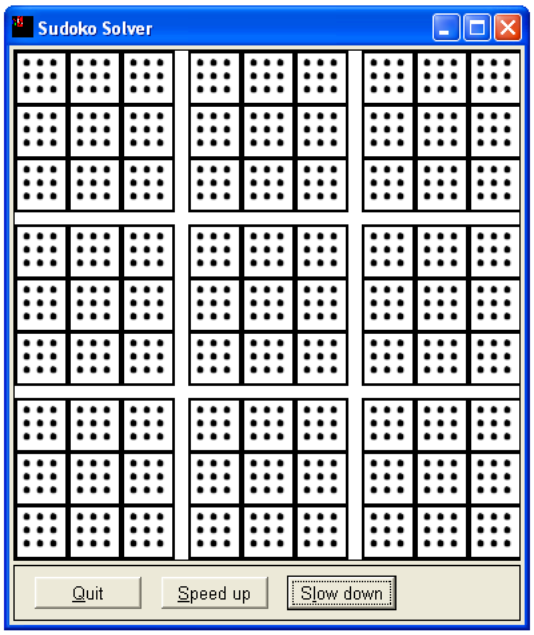

Figure 4.23: Initial numerical possibilities of the cells shown as black dots.

3. `init_data/0`: as previously mentioned at the beginning of this sub-section 4.2.5, this implementation initially sets a few cells to certain numerical values manually. The predicate `init_data/0` generally serves this purpose and consequently leads to the elimination of some of the possible values that other cells had available.

   To be precise, this predicate updates the values of a chosen set of cells into different numerical values. This is done algorithmically through throwing a constraint `cell/5` for each of the chosen cells to be updated. The first four arguments are the same as the constraint `cell/6`'s first four arguments and the fifth argument represents the numerical value the cell is set to. Regarding the visualisation, for each `cell/5` constraint thrown for a cell, a predicate `updatecell1` is called to update the visualisation.

   Since there is a `cell/6` constraint present in the store for every cell, for each `cell/5` constraint thrown into the store through the predicate `init_data` as previously explained, the following rule is fired:

   ```
   cell(A,B,C,D,_) \ cell(A,B,C,D,_,_) <=> true.
   ```

   A `cell/5` constraint indicates that this cell is set to a value, thus there is no need to have a `cell/7` constraint carrying the domain of this cell any more.

   The next step is performing the eliminations on the cells that do not have a numerical value it is set to yet. This is done through eliminating such value from the cell's:

   - same column:

```
cell(_,B,_,D,V) \ cell(A,B,C,D,N,L) <=> select(V,L,LL) |
            N1 is N-1, N1>0, updatecell1(A,B,C,D,N1,LL,red),
            cell(A,B,C,D,N1,LL).
```

- same row:

```
cell(A,_,C,_,V) \ cell(A,B,C,D,N,L) <=> select(V,L,LL) |
            N1 is N-1, N1>0, updatecell1(A,B,C,D,N1,LL,red),
            cell(A,B,C,D,N1,LL).
```

- and same box:

```
cell(A,B,_,_,V) \ cell(A,B,C,D,N,L) <=> select(V,L,LL) |
            N1 is N-1, N1>0, updatecell1(A,B,C,D,N1,LL,red),
            cell(A,B,C,D,N1,LL).
```

All three rules have the same idea on how the elimination takes place. The main difference is which position arguments to compare. The comparison is done through the head of each rule between a cell with a set value, let it be referred to as $C1$, and a cell $C2$ that is not yet bound to a value.

For example, checking that $C2$ is in the same column as $C1$ is done through checking if they have the same horizontal box reference B and the same horizontal position D in the box. If such conditions are satisfied, then the first rule, off the above rules, is fired. The numerical value V that $C1$ is set to, is removed from $C2$'s domain list L and a new updated list LL is output. Then, `updatecell1` is called for $C2$ with the list LL to update the visualisation. Also, since this is a simpagation rule, a new constraint `cell/6` is thrown into the constraint store with the new list LL to replace the one removed by the rule. The same process is run for the rest of the cells in the same row as $C1$ as well as any cell in the same column or box, through firing other two rules. This results in having the visualisation updated as in figure 4.24

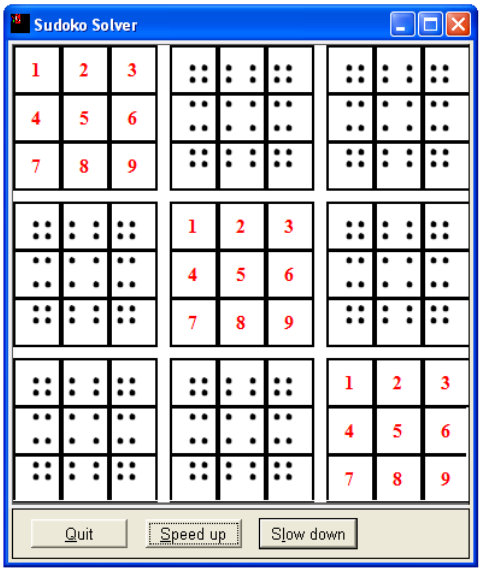

Figure 4.24: Some cells are set to numerical values thus causing eliminations in the domain of other cells.

The predicate `init_data/0` does not have any processes more to run after being done with throwing the `cell/5` constraints as previously mentioned, have them matched with rules causing them to fire, and no other constraints are available in the constraint store to match any rule. This leads to calling `solve/0`'s last predicate `try/0`.

4. `try`: this predicate throws a `fillone/1` constraint carrying the value 1.

```
fillone(N), cell(A,B,C,D,N2,L) <=> N2=N |
             member(V,L), updatecell1(A,B,C,D,1,[V],red),
             cell(A,B,C,D,V), fillone(1).

fillone(N) <=> N < 9 | N1 is N+1, fillone(N1).
```

The idea of using the constraint `fillone` through the above two rules is to search the constraint store for the cell with the smallest domain list. This is done through, first, firing the second rule above till the value N is equivalent to the fifth argument of any of the `cell/6` constraints available in the constraint store and thus firing the first rule. Once such a cell is found, its value is set to the first number of its domain list, as a trial, and the visualisation is updated accordingly such as in figure 4.25. Also, a cell/5 constraint is thrown to the store, thus causing the elimination rules mentioned before to fire. A new `fillone` constraint is fired as well at the end of the body of the rule. Thus, causing the same process of finding a cell to set a value to, to run. This leads to more eliminations performed and so on.

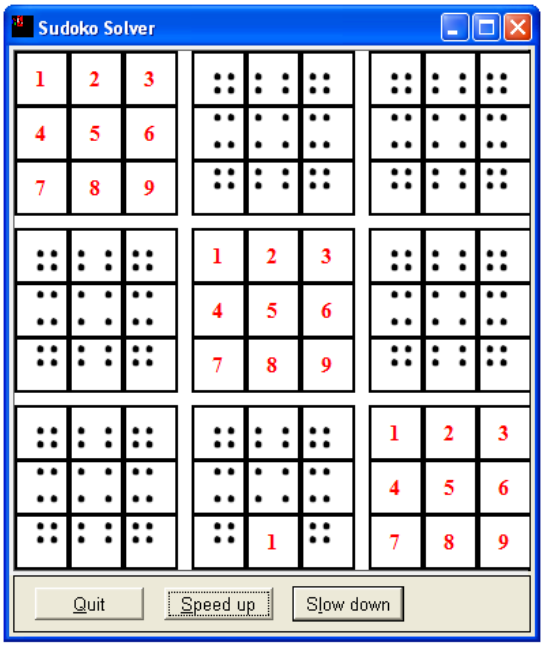

Figure 4.25: A trial value is set to a cell in order to search for a solution of the puzzle. Eliminations are done accordingly

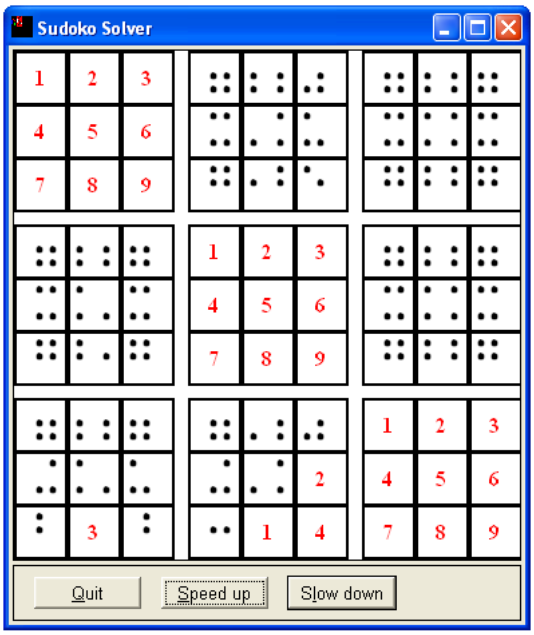

Figure 4.26: More eliminations performed as a result of setting values of more cells.

The eliminations are shown step-by-step through the visualisation until a solution is found to the puzzle as shown in figure 4.27.

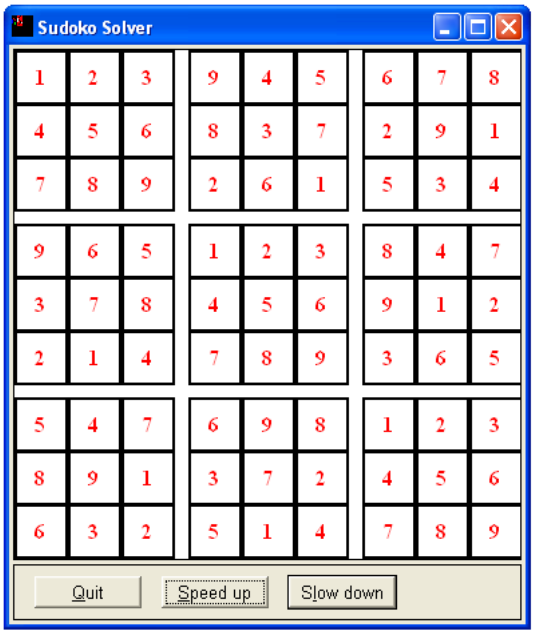

Figure 4.27: A solution of the Sudoko puzzle is reached.

An Extra Feature

In this work, an extra feature is added to the original implementation of the Sudoku problem, which is to give the user the choice to enter a personalized query with input cell values, i.e solving a personalised Sudoku puzzle. The algorithm then is run according to the values input by the user and the final solution is presented accordingly. An example of such query could be:

```
:- solve([cell(a,a,1,1,8), cell(a,a,1,2,3), cell(a,b,1,3,7),
   cell(a,c,1,3,4), cell(a,b,2,1,9), cell(a,b,2,3,4),
   cell(a,c,2,1,8), cell(a,c,2,2,2), cell(a,b,3,2,3),
   cell(b,a,1,1,2), cell(b,c,1,1,9), cell(b,c,1,3,7),
   cell(b,a,2,1,6), cell(b,a,2,3,7), cell(b,c,2,1,4),
   cell(b,c,2,3,8), cell(b,a,3,1,4), cell(b,a,3,3,3),
   cell(b,c,3,3,6), cell(c,b,1,2,1), cell(c,a,2,2,5),
   cell(c,a,2,3,8), cell(c,b,2,1,2), cell(c,b,2,3,3),
   cell(c,a,3,1,1), cell(c,b,3,1,8), cell(c,c,3,2,4),
   cell(c,c,3,3,5)]).
```

where the following implementation of `solve/1` is added to the code.

```
solve([H|T]):-  start, init_board, init_data([H|T]), try.
```

This predicate takes an input argument as a list of `cell/5` constraints specifying the cells to have their values set. The predicate `solve/1` works the same as `solve/0`, with the exception of the `init_data/1` predicate part.

```
init_data([]).
init_data([cell(A,B,C,D,Val)|T]):- updatecell1(A,B,C,D,1,[Val],black),
                                   cell(A,B,C,D,Val), init_data(T).
```

Previously, `init_data/0` was used to manually set values of a selected number of cells as part of the algorithm. However, the added predicate `init_data/1`, carries its argument as the list of `cell/5` constraints entered by the user and initially visualises the cells bound to those values and performs the necessary eliminations. This is shown in figure 4.28 where it is observed that the input values are shown in black. This was done through passing the last argument of the predicate `updatecell1/7` as black.

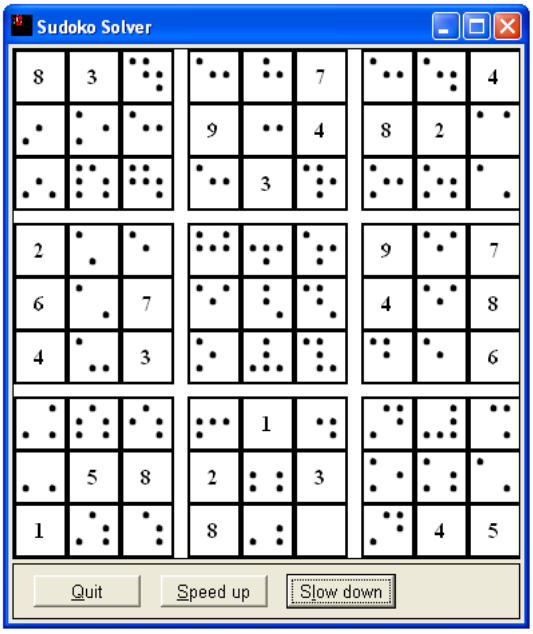

Figure 4.28: A personalised Sudoku puzzle where the input values are shown in black and eliminations are performed accordingly.

Then, the predicate `try/0` is run same as before. Using the personalised option of solving the Sudoko puzzle enables the user to see the difference between the input values, shown in black, and the resulted values from the eliminations, shown in red. The figures 4.29 and 4.30 show different stages of the visualisation.

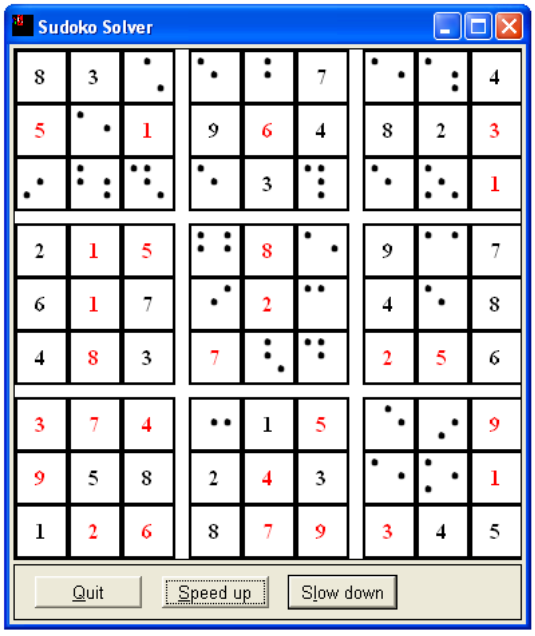

Figure 4.29: More values are set and more eliminations performed.

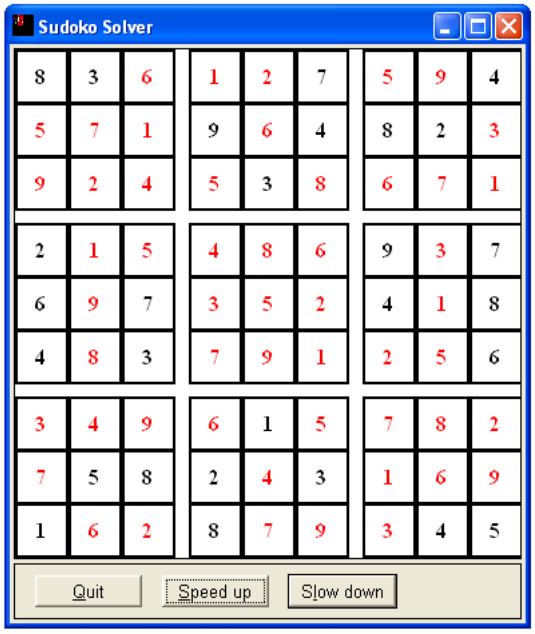

Figure 4.30: A solution is reached for the personalised Sudoku puzzle

Same as the N-Queens problem discussed previously in sub-section 4.2.4, one can use SWI-Prolog's ";" command for alternative solutions of the Sudoku puzzle.

4.3 Related Work

In [13], it was also of interest to visualise some CHR algorithms. Three main clusters of algorithms were chosen for that work; sorting, boolean algebra and searching algorithms. However, a different approach to accomplish the visualisation was taken; program transformation. This was done through a source-to-source transformation tool where the input CHR program is transferred into a special relational form. A Java application is then provided in order to parse the input file and extract the important information and triggers in the needed format. The user is requested to input some required information regarding the CHR code to serve some transformation purposes. Then according to the algorithm type chosen by the user, it applies the corresponding transformer depending on the chosen program. And finally, the visualisation is shown through the Java tool, where the connection between the solver and the tool is done through SWI-Prolog's JPL interface.

The main difference between the previously mentioned work and the work presented in this thesis is that there is no need to do any source transformation in the visualisations implemented for this project. Prolog, along with the graphical help of XPCE, is solely used with CHR to output the five algorithms' visualisations. This avoids any possible poor performance or clashes that can happen as a result of extensively accessing the different environments of multiple languages employed together to produce the output visualisation[8].

Chapter 5

Conclusion

5.1 Conclusion

Since visualising algorithms aids one's comprehension of how the computation takes place, it was appealing to this work to visualise CHR algorithms. In general, representing algorithms graphically has been a field of interest especially for the computer science world. It is essential to ensure a student's understanding of a newly introduced language or algorithm. This can be done through presenting the code of an algorithm accompanied by a visualisation that stimulates the eyes and help follow the computation of such algorithm graphically. For CHR algorithms, it is important to understand the concept behind how the constraints handling and how choices are committed. All of the previously mentioned factors have motivated this work greatly.

One of the main goals of this thesis, was to use only Prolog with CHR for the implementation part of this work, through SWI-Prolog acting as the compiler. However, as mentioned previously in chapter 3, Prolog does not own graphical capabilities and thus other options had to be explored. XPCE was found to be the most suitable graphical approach, since this portable tool-kit still uses only Prolog. Thus, it provides maximum stability without the need to employ and merge different language environments to serve the graphical aspect of this work.

In conclusion, this work presents visualisations of five different algorithms implemented in CHR and Prolog, with the help of XPCE for the graphical parts. Unique graphical representations were chosen for each of the two fundamental algorithms; finding the minimum of a set of numbers and calculating the N^{th} term of a Fibonacci series. While the other three algorithms; Wolfram's cellular automaton, N-queens and Sudoku problems, are all grid-based and thus their animations were implemented accordingly. Each visualisation shows a step-by-step workout of the algorithm that can aid the explanation of how CHR works and hence can be used for educational purposes.

5.2 Future Work

Concerning the future work, it can be divided into three main points of interest:

1. **Developing a Tool:**

It would be of more use to have one tool handling the algorithms all together, where a console can be embedded in the tool itself for the user to run the visualisations' codes from, instead of doing that externally through the SWI-Prolog console. Also, the aim would be to explore the possibility of developing such a tool using the same graphical approach, XPCE. Such a tool can be extended to include a wider collection of newly implemented algorithm visualisations. Another idea for the implementations added to the tool, is to generalise them as much as possible; i.e. make them semi-automatic. This can be done through providing certain tags for common actions or commands for example.

2. **Enhancing Speed Control:**

   Instead of using buttons that change the values of the delay to be made between the visualisation's steps, a more advanced approach can be explored. For example, having a speed slider controlling such an attribute to allow smoothness of the animation.

3. **More User Interaction:**

   The user can be given graphical options to choose from, thus allowing him to interact more with the to-be-developed tool. Different options on how a visualisation is presented can be added for the user to choose from. Also, investigating the possibility of enabling the user to rewind to a certain point of the visualisation or skip to a next step.

Bibliography

[1] R. Baecker, "Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science," *Software Visualization: Programming as a Multimedia Experience*, 1998.

[2] G. Rössling and B. Freisleben, "ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation," *Journal of Visual Languages & Computing*, vol. 13, no. 3, 2002.

[3] T. Frühwirth, "Introducing Simplication Rules," *Technical Report ECRC-LP-63, European Computer-Industry Research Centre, Munich, Germany*, October 1991.

[4] T. Frühwirth, *Constraint Handling Rules*. Cambridge University Press, 2009.

[5] SWI-Prolog 6.1.6, "http://www.swi-prolog.org," Accessed: March 2012.

[6] B. Bemoen and J. Sneyers, "Optimizing Compilation and Computational Complexity of Constraint Handling Rules," 2008.

[7] S. Wolfram, *A New Kind of Science*. Wolfram Media, 2002.

[8] E. W. Weisstein, "Rule 110. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Rule110.html," Accessed: April 2012.

[9] E. W. Weisstein, "Queens Problem. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/QueensProblem.html," Accessed: May 2012.

[10] E. J. Pegg and E. W. Weisstein, "Sudoku. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Sudoku.html," Accessed: May 2012.

[11] A. Kerren and J. Stasko, "Chapter 1 Algorithm Animation," *Software Visualization*, 2002.

[12] S. Abdennadher and M. Saft, "A visualization tool for constraint handling rules," in *In Proceedings of 11th Workshop on Logic Programming Environments*, Citeseer, 2001.

[13] S. Abdennadher and N. Sharaf, "Program Transformation for Visualizing Algorithms written in Constraint Handling Rules," in *22nd International Symposium on Logic-based Program Synthesis and Transformation*, LOPSTR 2012, in press.

[14] "CMU Artificial Intelligence Repositry. http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/gui/xwip/0.html," Accessed: June 2012.

[15] J. Wielemaker and A. Anjewierden, "An Architecture for Making Object-Oriented Systems Available from Prolog," *Social Science Informatics (SWI), University of Amsterdam*, 2 August 2002.

[16] T. Schrijvers, "ICLP 2008. Constraint Handling Rules - A Tutorial for (Prolog) Programmers. http://people.cs.kuleuven.be/~tom.schrijvers/Research/papers/tutorial_iclp2008.pdf," 9-13 December, 2008.

[17] M. Kaeser and T. Frühwirth, "CHR Online. http://chr.informatik.uni-ulm.de/~webchr/," Accessed: March 2012.