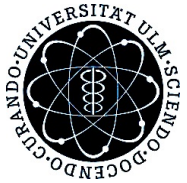
German University in Cairo
Faculty of Media Engineering and Technology
Computer Science Department



Ulm University
Institute of Software Engineering and Compiler Construction

Animation of Mathematical and Graph-based Algorithms expressed in CHR

Bachelor Thesis

Author:	Mostafa Ali Said
Supervisor:	Prof. Dr. 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 acknowledgement has been made in the text to all other material used.

Mostafa Ali Said
1 July, 2012

Acknowledgements

This thesis would not have been possible if not for the encouragement, the assistance and continuous support of the people around me. As I can not name everyone, only some of those worthy of acknowledgements were particularly mentioned.

First and foremost, I would like to thank my family, my parents, my sister, my brother and my newly born nephew whom i still did not get the chance to see, for always being there, for supporting and believing in me.

I would also like to show my deepest gratitude towards both my supervisors, Prof. Dr. Thom Frühwirth for allowing me to do this project and for his guidance and Amira Zaki, as she has made available her support in a number of ways. She was always dedicated to offer help whether technical or not. She never stopped the moral support as well throughout the work.

I am indebted to a number of colleagues who helped me in this work and what lead to it distinctly my colleagues who accompanied me here in Ulm and my colleagues in the Media Engineering and Technology department in the German University in Cairo.

Finally, I would like to thank all my friends and those who helped me become the person I am today.

Abstract

Algorithm Animation is a subtopic of Software visualization, this branch focuses on the development process of software and the visualization of its related artifacts, while satisfying the goal of the possibility of its usage for educational purposes Constraint Handling Rules (CHR) is a concurrent committed choice rule based declarative programming language. This work creates visualization for CHR programs using XPCE/Prolog whilst mentioning the process of the creation from the start to create a framework as a basis for developing any algorithm visualization

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The renaissance of Rule-based programming is the outcome of its recurring usage in various areas such as Business Rules, Semantic Web, Computational Biology, Verification and Security.

The ability to offer a simple option to build scalable and distributed systems leads to the constant development, improvement and demand of languages such as Constraint Handling Rules.

Constraint Handling Rules (CHR) is a concurrent committed-choice constraint logic programming language that was motivated by the inference rules which are commonly used in Computer Science.

The amount of constraints added during the execution of a CHR program could make it cumbersome to trace or go through an algorithm step by step while imagining how it changes the initial data set, this is why a visualization and animation of algorithms could be of utter importance for CHR. Hence, this project ' Animation of Mathematical and Graph-based Algorithms expressed in CHR' became an interesting and beneficial topic in the scope of CHR.

Software Visualization is a broad branch of computer science, it is the general term that contains topics like Algorithm Visualization or animation, program visualization, visual debugging and data structure display amongst others. This field has witnessed massive interest and growth in recent years due to its importance to everyone in the computer industry and academics teaching or learning.

1.2 Aim of the Project

The aim behind this project is to have a visualization of algorithms, where CHR constraints are represented as graphical objects on the screen. The states of these objects change with each significant event, thus allowing the user to see what happens during the algorithm execution step by step. This would not only facilitate envisioning large scale and complicated algorithms; but it would also be beneficial to use as an educational tool for students learning CHR. The project aims at creating a basis for a generic visualization tool, this is achieved by defining a framework for the visualization of any algorithm and using this framework in developing animation for six different CHR programs as a proof of concept.

That is why the visualization of CHR programs contributes to the expressiveness of CHR due to the fact that it adds another dimension to the imagination of constraints and how they are formulated and exhaustively applied.

1.3 Thesis Outline

This thesis includes four other chapters. After an introduction of the topic is given, the background chapter will give a brief insight into Constraint Handling Rules and the idea of algorithm visualization, its history and similar terms and concepts will be discussed. The theoretical background of the thesis will be represented, as well. Chapter 3 contains the general approach, the decisions taken for the implementation, the steps to be considered in the implementation and the algorithm categorization. After that the implementation, the examples is discussed in chapter 4. Finally, the results will be analysed for reference and the possibilities for future work are suggested.

Chapter 2

Background

2.1 Software Visualization

Software by default is not visible, it is just a precise description, expressed in a computer programming language. Thus with the software evolution and the software demands and requirements growing exponentially, complex programs were created to satisfy the needs of the people. These complex programs could reach millions of lines of code, which requires something special to help explain them, as a mere textual display will not be enough. This paved the way for the field of software visualization

Software Visualization(SV) is the process of using computer graphics and animation to portray, represent and depict computer programs, processes, data structures and algorithms.

Software Visualization is a broad branch of computer science, it is the general term that contains topics like Algorithm Visualization or animation, program visualization, visual debugging and data structure display amongst others.

As one of the branches under SV, algorithm animation (AA) is a form of visualization where the focus is on visualizing the execution or behaviour of the algorithm without paying much attention to the other aspects of the algorithm. Another term widely used is Algorithm Visualization (AV), which usually stands for a slightly larger scope, but throughout this work both terms will be addressed interchangeably to refer to the definition of Algorithm Animation.

2.1.1 History

As is the case with anything in history, the exact time where software visualization research was started is not quite known. However, in 1966 Ken Knowlton created something that would later be considered an algorithm animation, he made a short 30 minute black and white film about an example list processing with L6 programming [4]. Later in 1974, Hopgod created a set of animations in form of short films on hash tables. Then in 1975, Baecker presented two systems that made it possible for an instructor to produce short quick-and-dirty single-concept film clips with only hours of effort [5].

Sorting out Sorting

Many people are lead to believe that this short film was the start of algorithm animation research due to the revolution it created in the field and the fact that sorting algorithm are still visualized in the same manner as no other representation for the structure nor behaviour has given the same results . 'Sorting out Sorting' is an animation created in 1981. It is a 30 minute short sound colour film and one of the cornerstones of Algorithm Visualization research. The film portrayed the animation of 9 different sorting algorithms, the elements were represented in form of coloured bars stationed horizontally where each bar represents the element of the array while its height depends on the value of the element itself. Thus the structure was first visualized followed by the animation the behaviour of the internal sorting. The film also includes a comparison between the time it takes to visualize different sorting algorithms. Figures 2.1 and 2.2 show parts of the film.

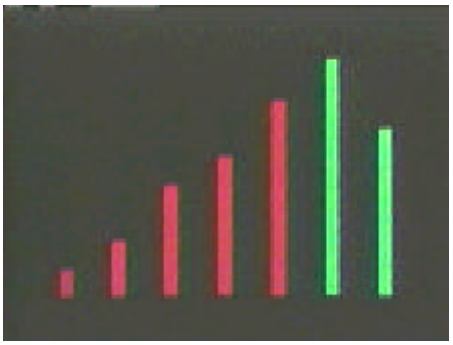


Figure 2.1: A part of the sorting, Courtesy [1]

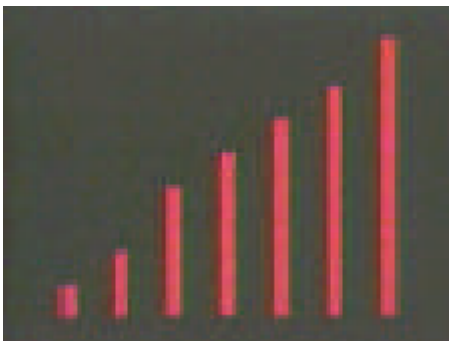


Figure 2.2: After the List is sorted, Courtesy [1]

2.1.2 Previous Algorithm Visualization Systems

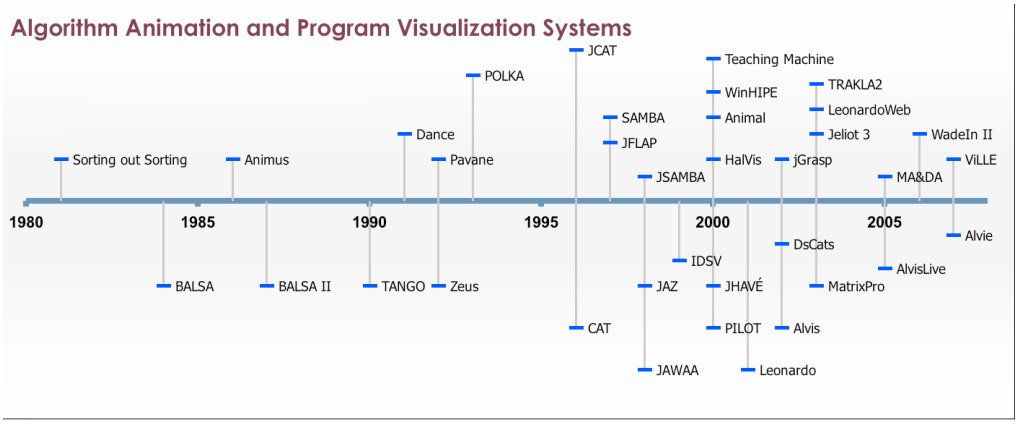


Figure 2.3: Timeline of Famous Algorithm Animation Systems, Courtesy [2]

The above picture shows a timeline of algorithm visualizations starting from the 'Sorting out Sorting' video, the figure shows an exponential increase after the year 1995 which is the year Java appeared.

The most notable AV systems that were considered as milestone in the field since the 'Sorting out Sorting' video are: Brown Algorithm Simulator and Animator (BALSA) which introduced the interesting events paradigm, TANGO(Transition-based ANimation GeneratiOn) which introduced the path transition paradigm [6], XTANGO the following edition of TANGO. Zeus was the first system to use colour and sounds.

POLKA (Parallel program-focused Object-Oriented Low Key Animation), Swan, ANIMAL (A New Interactive Modeler for Animations in Lectures) [7], JAWAA (Java And Web-based Algorithm Animation) [8], JHAVE (Java-Hosted Algorithm Visualization Environment)[9], and AlViE 3.0 are some of the other significant systems.

2.1.3 Importance of Algorithm Visualization

Algorithms visualization is very important to the field of computer science, as it is used by most people of the field. For researchers in the field of AV, using new concepts and applying learning and perception concepts to create new visualization techniques and build better visualization systems could become a challenge to themselves.

For practitioners, AV can help in the process of designing and debugging algorithms. Also, a stable animation could help establish an abstract comparison between the running times of two algorithms.

Finally for computer science students, AV has immense pedagogical potential as it could be used as a primary learning source for concepts or a primary learning source for algorithm or data structure operational details thus helping students understand algorithms faster and in greater depth. A great example of how effective software visualization is, even in its simplest forms, would be the use of box to describe variables in computer science text books; or using a column of boxes to represent arrays or an arrow to represent a pointer. All these are things we take for granted although they represent a primitive type of software visualization.

2.1.4 The Pedagogical application of Algorithm Visualization

Despite the noticeable increase of algorithm visualization systems, their use in computer science education did not cause the same revolution and does not share the same growth. This is due to the fact that the pedagogical requirements that are needed for effective educational use are not addressed in most of the animations and visualization systems. Instead, programmers tend to focus their attention on the graphics rather than the pedagogy, which will not enhance the learning experience no matter how good the graphics are. if the animation is only used as a passive video on the algorithm operation the student will not feel involved nor will he understand better the behaviour of the algorithms.

However, some features could render an animation, even with the simplest graphics, a great educational tool since these features would encourage students to interact with the AV system.

For these reasons and the lack of Algorithm visualization systems that can be used as effective educational tool, Rößling and Naps[10] examined AV systems and students' reaction to them and decided upon the most

important pedagogical requirements an algorithm visualization needs to satisfy to be considered a useful educational tool.

These requirements are:

Reliably reaching a large target audience: It should be platform independent, that way it is not meant for a certain target audience. In case it is not possible to make it platform independent then the platform that would form and satisfy the largest target audience being chosen.

Moreover, allowing users to provide input to the algorithm: the student should be more than merely a spectator, it should be possible to provide different input to see how the program and the animation would differ for different inputs.

The third requirement is smooth motion: the changes that happen should have a sufficient amount of time between them for the user to be able to process the previous action or change.

Rewind capability: A rewind capability is considered very useful in an AV system, since the student can miss an action or get confused in the middle of the animation, thus the rewind capability could help the student understand by going back to the exact second of the animation that started the confusion without having to repeat the whole animation.

General-purpose systems: The system should not be topic specific, if realising that was not possible, then it should ensure covering the most fundamental topics.

Hypertext explanations of the visual display: A few algorithms could have certain functionalities that could be hard to understand and the student might find trouble absorbing what is happening in the animation even after repetition. Thus a hypertext explanation of the actions or the event with the events themselves is beneficial as well.

Structural view of algorithm: The algorithm main parts should be stressed upon.

Interactive prediction: The system could have a few stop and think questions, which pop up before important events asking the student to predict the next step. This is one of the requirements that were proved to have a somehow neutral effect on he students' learning experience since the student could treat it as merely a guessing game. Hence its importance as a pedagogical requirement is minimal.

Integration with database for course management reasons: An addition that might help with the interactive prediction, basically a database is added where the student's answers are kept.

2.1.5 State of the Field

In 2010, Shaffer et al. [3] made their second report on the state of the field of Algorithm Visualization, after the first one done in 2007 [11], where different aspects of the state of the field of Algorithm Visualization are discussed.

During the work of the first report, they created the AlgoViz Wiki in which they collected links to most of the algorithm animations and visualization systems. The Wiki also contains a bibliography of Algorithm Visualization related research. After collecting these visualizations systems and animation, the wiki had become a repository that can be used in many ways one of which is to make a case study on the topic.

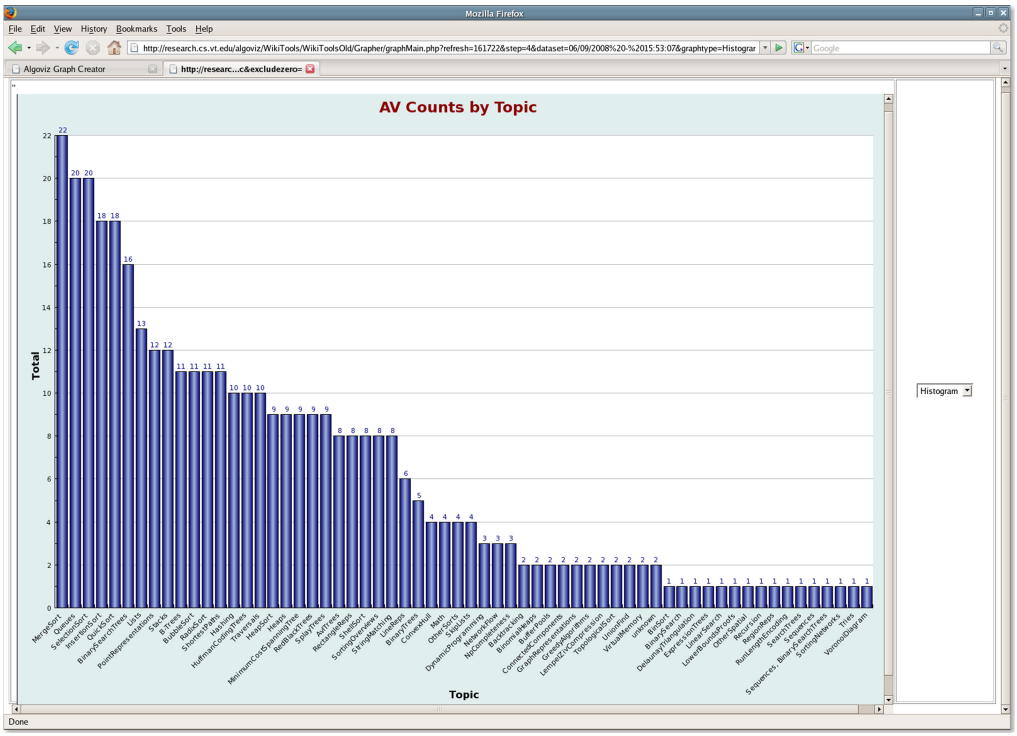


Figure 2.4: AV counts by topics, Courtesy [3]

The available visualization

The topics covered in visualization collected in the AlgoViz Wiki were clustered according to the type of the algorithm and its behaviour. Afterwards, a count by topics was made which showed the results expressed in figure 2.4. The sorting algorithms came on top with the most visualization, they were followed by tree traversal algorithms as these topics are the hardest to understand from merely the implementation.

2.2 Constraint Handling Rules

2.2.1 Introduction

Constraint Handling Rules (CHR)[12] is a concurrent committed-choice constraint logic programming language that was motivated by the inference rules which are commonly used in Computer Science. It was invented in 1991 by Thom Frühwirth. CHR does not necessarily impose itself as a new programming language, but as a language extension to a host language. Those that can be used at the moment are: Prolog, Haskell, Lisp, Java or C. It comprises guarded rules that transform multi-sets of atomic formulae (constraints) until exhaustion in a similar way to automated theorem proving.

The language was originally designed for writing user defined constraint solvers, now it is recognized as an elegant general purpose language, it combines elements of Constraint Logic Programming and term rewriting analysis.

The clean logic-based semantics of CHR ensures that several desirable properties hold for CHR programs and also facilitates non-trivial program

2.2.2 Syntax

A CHR program consists of a finite set of rules. There are three main types of rules:

- Simplification Rule: replaces constraints with simpler, logically equivalent constraints.
- Propagation Rule: adds constraints which may be logically redundant but enable further simplification.
- Simpagation Rule: keeps a part of the head in the constraint store and removes another

Each rule consists of:

- A head is a non-empty sequence of CHR constraints.
- A guard is a sequence of built-in constraints, it is a precondition on the applicability of the rule.
- A body is a sequence of CHR and built-in constraints.

2.2.3 Simplification Rules

$H_1, ..., H_n \Leftrightarrow G_1, ..., G_n | B_1, ..., B_n.$

In this rule, the constraints in the guard are checked, if they matched, then the body is executed and all the constraints in the head are removed and replaced in the constraint store by the constraints in the body

2.2.4 Propagation Rules

$H_1, ..., H_n \Rightarrow G_1, ..., G_n | B_1, ..., B_n.$

In this rule, the constraints in the guard are checked, if they matched, then the body is executed and all the constraints in the body are added to those in the head in the constraint store.

2.2.5 Simpagation Rules

$H_1^{kept}, ..., H_n^{kept} \backslash H_1^{removed}, ..., H_n^{removed} \Leftrightarrow G_1, ..., G_n | B_1, ..., B_n$

It is a combination of Simplification and Propagation rules. In this rule, the constraints in the guard are checked, if they matched, then the body is executed and all the constraints that are in the kept part of the head are kept in the constraint store, the others which exist in the removed part of the head are removed

2.2.6 Confluence

A CHR program is confluent, if for any given goal, in any order of rules application, the same result with the exact same final state is yielded.

2.3 JPL: Java Prolog

JPL [13] is an interface that links between Java and Prolog that consists of a collection of Java classes and C functions. The Java Native Interface(JNI) is responsible for the connection to a Prolog engine using the Prolog Foreign Language Interface(FLI). This interface is currently being standardized in different implementations of Prolog. JPL takes advantage of native implementations of Prolog on supported platforms, making JPL more than just a pure java implementation of Prolog.

Currently, JPL only supports the embedding of a Prolog engine within the Java VM. Future versions may support the embedding of a Java VM within Prolog, that would allow us to make use of the rich class structure of the Java environment from within Prolog.

JPL exists in two layers, both a low and a high level interfaces, for C programmers who may wish to port their C implementations, the low level is provided since it is for the Prolog FLI that would be used during the porting, it is also highly SWI-Prolog specific, that is why the current version of JPL only works on SWI-Prolog .The high level is provided for those who do not care about the details of the Prolog FLI such as the Java developer.

There are two options that can be used to work with JPL: Working with Java from Prolog or Prolog from Java.

2.4 XPCE

XPCE is an object-oriented toolkit for developing Graphical User Interfaces (GUIs). XPCE can easily be connected to a new language due to the fact that the interface between it and the host language is very small, however the library's greatest accordance is with languages that are symbolic strongly typed and dynamically typed languages considering that XPCE itself is dynamically typed.

For a dynamically typed host language such as Prolog or Lisp, the type of the host-language construct passed is determined by the interface and translated into the corresponding XPCE object the interface determines and translates it into the corresponding XPCE object.

The implementation of all of XPCE on top of its primitive graphical guarantees there are no platform-specific limitations in the manipulation and semantics of certain controllers. Tested Unix platforms which saw XPCE running successfully include SunOs, Solaris, AIX, HPUX, IRIX, OSF/1 and Linux.

The language on which XPCE is used mainly are C++, Lisp and Prolog. However, Prolog is considered the main target language.

2.4.1 Syntax

Controlling XPCE from Prolog [14] is accomplished using four primary predicates. These predicates allow the creation, manipulating, querying

and destroying of the objects which form the basic entities of XPCE. The four predicates are `new/3, get/3, send/2, free/1` . A brief description on the syntax of XPCE/Prolog is given, starting with each of the four predicates.

- `new/2`: this predicate handles the creation of XPCE objects, it creates an XPCE object and gives it a reference which represents an identification for this object. `new(Reference,NewTerm)` this would create a new object of the type $NewTerm$ and give it the reference $Reference$ e.g. `new(P,`point(10,20)): creates a new point and assigns a reference to the variable P.

- `send/2`: this predicate handles the modification of XPCE objects, it has two arguments, the first is a reference to the object to be modified, the second is a term which is the name of the method to be invoked on the object. The predicate `send` however can consist of more than two arguments, it could go up to 12 arguments, the 10 extra arguments are considered optional parameters, they could be appended or added to the predicate but in case they are omitted then the execution continues after setting them to the default value. e.g. `send(@demo,`open) . This query invokes the method open on the object with the reference @*demo*

- `get/3`: this predicate handles the querying of XPCE objects. It extracts information about the object. The predicate consists of three arguments, the first two are exactly the same as `send/2`, the third argument is the variable that is set to the return value. This value is a reference unless it is an XPCE name object then the value could be returned as a Prolog atom, XPCE integer and XPCE real objects. e.g This query `get(@demo,display,D).` would unify D to `@display/display`. On the other hand, this query `get(@772024,y,Y).` would unify Y to 20

- `free/1`: responsible for the destroying of the XPCE objects. It has only one argument which is the reference of the object to be removed. e.g. `free(@demo).` This would remove the dialog object window from the XPCE object base, it also removes the associated window form the screen.

Chapter 3

Approach

To create the visualization of the CHR problems, there are several things that should be taken into consideration:

1. The issue of choosing which of the available host languages to use for the CHR implementation.
2. The selection of the approach to be taken in terms of which programming language to use for the graphical representation.
3. Imagining and visualizing how classical CHR algorithms would be represented as graphical objects how would they change with each event.
4. Based on the visualization of the classic problems, a classification of the algorithm in term of visual representation should be reached CHR problems should be distinguished in clusters and the consideration of the possibility of adding other categories.
5. The task of contemplating the depiction of the tool itself, its buttons and its options that would be available to the user for the user to change during the visualization.

3.1 Host Language

The work started by examining the tools to be used for the Implementation: First,choosing which host language to use for the CHR implementation, the options were: Prolog, Java, Haskell and C.

Due to the fact that JCHR(Java CHR) is not stable and no longer supported enough for the basis of creating a project that would be worked on in the future, the option Java and using JCHR was disregarded.

CCHR is the CHR system embedded in C, the system is described in [15] as the best and fastest implementation of CHR. However, the system is relatively new and the primary problem faced with C would be the platform specificness. As a system in C would not be platform independent, which violates the first pedagogical requirement mentioned in chapter 2, for these reasons C was disfavoured.

Finally Prolog was chosen over Haskell, since it is the most popular host language for CHR, it is the host language mostly used in research about CHR, in the development of several projects with CHR and most notably because it is the host language used in the education of CHR.
Afterwards, a quick comparison between Prolog implementations lead to favourite SWI Prolog over the other implementations most notably Sicstus Prolog due to the fact that SWI has better Web related features and has a free open source license thus allowing for maximum target audience who could be demotivated by the necessity of buying a license for Sicstus.

3.2 Visualization Language

Settling down on the first decision and choosing Prolog as a host language would lead to the second approach decision, it is the choice of the language that would be used for the visualization. Since SWI-Prolog has no graphical capabilities there were different approaches:

One popular approach would be to use an external language that has good graphical user interface development, such as Java and Visual Basic, and then link the CHR program to this external language. Popular options of this approach are: Tk/Tcl, JPL/Java, Visual Basic and Delphi.

Another popular option would be to use XPCE as the graphical language, since XPCE is a powerful toolkit that has been created for GUI development in Prolog from the start.

The last possible approach would be the direct access to graphical Application Programming Interface(API): Xwip is an important example of this class of GUI approach for Prolog. Problem is that most of these API's are fairly low-level and a lot of work is required to get the data types of the API properly and naturally represented in Prolog.

The path taken was to try two approaches since they both have their advantages and strength points

- Java as the external language for the user interface while using JPL to link between the front end and the back end. Since Java has different GUI libraries and has great capabilities of representing different primitives that could be manipulated easily.
- XPCE has been developed for GUI development in Prolog from the start, since it was used in the Prolog graphical tracer, and no major complications can occur as the XPCE code would be embedded into the algorithm's source code.

3.3 Categorization of Algorithms' Visualizations

After examining most of the classic computer science algorithms in CHR implementation, it became evident that these algorithms and programs should be split into these categories which would cover:

- Grid based, e.g., Prime, Sudoku, Zebra problem, Board games.
- Graph based (including tree representations) e.g DFS, shortest path, transitive closure and Dijkstra.
- Chart based whether it is bar chart, pie chart and mathematical graph e.g., sorting algorithms, calculation based problems such as accumulated sum, factorial and Fibonacci.
- Bitmap based problems e.g. programs that must be represented with certain images.

This clustering aided the decision of which algorithms to implement, as the clustering covered all the visualization possibilities. Their characteristics would also best suit most of the algorithms data and define the structure behaviour and evolution of execution. Hence, the algorithms chosen were: Prime sieve, Euclidean Algorithm, Ordered Merge Sorting, Exchange Sort, Depth First Search and Ballistic trajectory simulation.

3.4 Preparation

During the work of this thesis, before visualizing any algorithm there is a preparation phase. This preparation phase included a few steps, procedures and considerations that needed to be addressed before starting with the implementation of the animation routines. These steps could guide any developer who is interested in developing a new algorithm visualization for any topic and could be used in the generalization of the visualization and the creation of the tool.

First, the algorithm itself should be implemented in CHR. Afterwards, categorising the algorithm and examining which visual cluster it should belong to must be considered. Subsequently the general graphical representation of the algorithm is imagined: thinking of every way the algorithm could be visualized in, after imagining the execution and behaviour of the algorithm.

The third step would be weighing in the options the advantages and disadvantages of each representation and visual categorisation and settling on the one that guarantees the best demonstration of the CHR code while presenting it properly. Then we investigate the types of primitives that would be used to represent the entities of the algorithm.

The fifth step is rather important which is defining the crucial events and operation and distinct them from the events that do not have an effect on the behaviour of the algorithm. The last two steps are: Settling on the animation routines and actions initiated by the significant events chosen in the previous step and thinking of how to handle extreme cases and how the visualization would handle them as well.

3.5 CHR Events

On the grounds that one of the steps of the preparation to visualizing any algorithm is to define its crucial events, CHR syntax and behaviour should be examined to be able to define all the events and states that can occur in a CHR program. These events should be listed to have a repository of events possible to occur.
When the list of possible events is in hand, we could understand the algorithm to be visualized, state its crucial events and accompany them with a defined animation routine that best highlights the behaviour. The CHR

syntax explained in chapter 2 helps extracting the events that could occur during the execution of an algorithm.
The possible events that can occur in a CHR program most notably include: A constraint is activated, deactivated, killed, or added to the constraint store, a variable is bound, two variables are unified, a guard is matched, a rule is fired or a body is executed.

As a general approach, while implementing each visualization, the effort was made to try to satisfy most of the pedagogical requirements discussed in section 2.1 as much as possible to maintain the possibility of using each visualization as an educational tool.

Chapter 4

Implementation

The choices of the algorithms to implement or visualize were made after a thorough consideration, a research on the state of the AV field in general and the state of the CHR visualization as well. The choice were made in attempt to cover most of the clustering, to show an implementation and animation to each category.
It was taken into account as well the implementations that are somehow complicated to understand or imagine since their visualization would help tremendously with their use and understanding.

4.1 Algorithms Implemented Using JPL

4.1.1 Primes

Prime sieve is considered one of the classical problems of CHR, it represents a grid based visualization.

The example was implemented while using Java from Prolog

The CHR implementation of the algorithm consists of two rules

```
upto(N) <=> N>1 | M is N-1, upto(M), prime(N).
prime(X) \ prime(Y) <=> Y mod X =:= 0 | true.
```

The first rule generates all the numbers from the selected number and stops before the number 1. The second rule handles the elimination of the non prime numbers by comparing any number with all the smaller numbers and then removing that larger number if the result of the modulus operation is zero.

The visualization code was added to these two rules to create the visualization. The input is entered as an `upto(N)`. This simplifies to the Prolog predicate, that creates the frame and initializes the grid with its size, and another constraint `upto/4` after adding the frame, content pane and the grid created in the initialization. The `upto/4` starts the generation and therefore the animation by firing the below rule.

```
upto(N,Grid,CS,F)<=> N>1| M is N-1, sleep(1),
                          addNumber(N,Grid,F),
                          upto(M,Grid,CS,F), prime(N,Grid,CS,F).
```

This rule adds the current number to the grid, adds its corresponding `prime/4` constraint to the constraint store after adding the constraint `upto/4` of the next number. The number is added in its designated index in the grid by calculating its corresponding row and column. This rule would fill the grid with the generated number with a delay between each addition, when all the numbers are added, the below rule will keep firing and executing its body until it eliminates all the non prime numbers.

```
prime(X,Table,Text,F) \ prime(Y,Table,Text,F)
                            <=> Y mod X =:= 0 |
                                atom_number(XStr,X),
                                sleep(1), remove(Y,Table,F).
```

If the number is a non prime number then it is removed from the grid. By the end of the eliminations the grid consists of empty cells and cells that contain the prime numbers.

Figure 4.1 shows the visualization of the program.Figure 4.2 shows that the visualization worked on large values with the same competence.

The addition of a few animation routines and functionalities were considered even simple text colouring and indices highlighting, however due to the limitations of working with Java from Prolog , they were not achieved. This lead to continuing the work with XPCE as the conclusion was it would be more beneficial to use XPCE for an algorithm animation of a CHR program because to perform animation routines at a step by step basis we need to work from Prolog and the abilities at hand while working Java from Prolog are rather limited eliminating many graphical advantages of Java and also JPL only works with SWI-Prolog.

The code used for the implementation is included in the appendix A, since the implementation is not explained in detail.

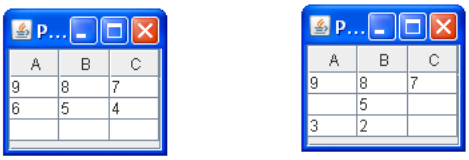


Figure 4.1: Primes

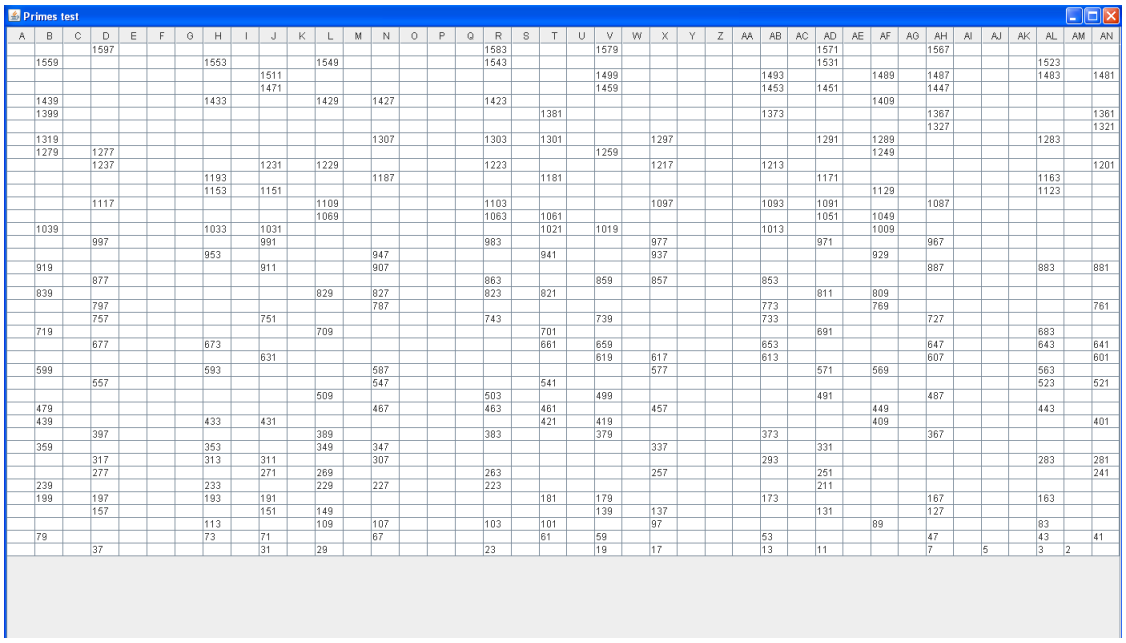


Figure 4.2: Primes big range

4.2 Algorithms Implemented Using XPCE

4.2.1 The Euclidean Algorithm

An algorithm that is used to find the greatest common divisor(GCD), which is one of the classical computer programming problems. The euclidean algorithm takes as an input two integers and outputs their GCD. This is achieved by always dividing the larger number by the smaller and then assigning the remainder to the variable holding the larger number. This is done repeatedly till the remainder is 0, which would mean reaching the GCD.

The CHR implementation has an advantage over other implementations of the euclidean algorithm since the number of inputs is not just restricted to two as it is the case in the implementation of other languages. The CHR

implementation has no limits over the number of inputs to its euclidean algorithm's implementation.

The two rules below for the implementation of the algorithm in CHR. The first rule is responsible for the clean up, the second handles the calculation of the Greatest common divisor.

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=> 0 < N, N =< M | L is M mod N, gcd(L).
```

However, in order to visualize this algorithm, a few predicates and constraints have been added to the implementation with some adjustments to the above rules as well.

The structure used for this algorithm was a bar chart as well, since this too had numbers or values that are being compared against each other and an action occurs according to the result of the comparison.

First to start the animation, the constraint `max(maxValue)`has to be queried by the user. This constraint has only one argument which is the variable that holds the largest value amongst the number entered as input to the euclidean algorithm. This should be followed by the list of numbers that consist of at least 2 that would be entered in the form of the constraint `gcd(Number)`.

The constraint `max/1` is just used for the initialization, however it has no real significance in the algorithm nor the visualization. It is merely considered a dummy constraint that simplifies to the predicate that initializes the scene.

```
max(X) <=> initialize_screen(X).
```

Subsequently, the initialization which includes nothing except the creation of the picture, which is a type of window, is executed.

Then each two `gcd/1` constraints entered cause the firing of the main rule of the algorithm.

The main rule that is responsible for the calculation itself and the animation is fired whenever two `gcd/1` constraints are in the constraint store.

```
gcd(N) \ gcd(M) <=>
              sleep(1),
              0 < N, N =< M,
              Top is M + 5 |
              free(@chart),
              new(@chart,
                 bar_chart(vertical, 0, Top, 500, 10)),
              send(@chart,append,new(BM,bar(M,M,blue))),
              send(@chart,append,new(BN,bar(N,N,blue))),
              send(@window,display,@chart),
              send(@window,flush),
              sleep(1) ,
              L is M mod N,
              send(@chart,clear),
              send(@chart,append,new(BN1,bar(N,N,green))),
              send(@chart,append,new(BL,bar(L,L,green))),
              send(@window,flush),
              gcd(L).
```

In the animation, the chart is created from the start with every application of the above rule because with large numbers that lead to small GCD would have the bars with a very small height eventually for the chart would have been initialized with a big maximum and range as well.

In the rule, it starts a delay of one second, created by the Prolog predicate `sleep/1` which stops the execution for the specified amount of time, for the user to have sufficient time to absorb the change and for the algorithm to have smooth motion in the animation, then the guard checks if both of the numbers are greater than 0.

If the guard matched then the chart is first freed to remove the last chart from the picture, then the new chart is created and added to the picture after the two bars representing the numbers are subsequently added and appended to it. A repaint `send(@window,flush),` is called to show the changes.
After the repaint, a delay of one second is added, the modulus is calculated, the chart is cleared and the result, which is the two constraints, the one that remained from the simpagation and the other created after the modulus operation, are added to the bar chart. Finally another repaint is called to show the result of the calculation.

This sequence continues until there is only one non zero `gcd/1` constraint left which would correspond to the output which is the GCD of the input.

The figure below shows part of the animation for the input: `max(94017),gcd(94017),gcd(1155),gcd(2035).`

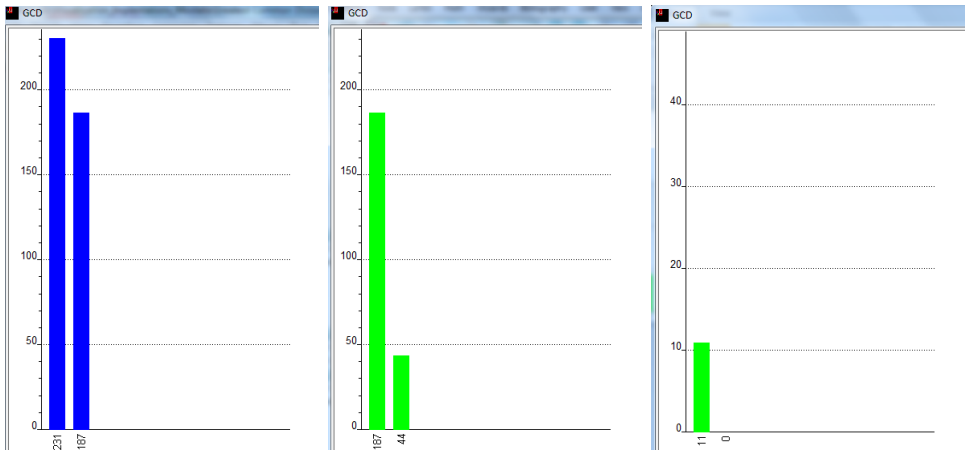


Figure 4.3: Greatest Common Divisor

4.2.2 Ordered Merging Sorting

As shown in figure 2.4, sorting algorithms are the most covered topics in algorithm visualization, they are always the ones that get the attention on the grounds that they are diverse, have a huge popularity amongst those in the field of computer science and the fact that due to their diversity, some sorting algorithm are complex to the point that their internal sorting needs visualization for them to become understandable for the user. For these reasons, there had to be some visualization for sorting algorithms in CHR.

This sorting algorithm is somehow unfamiliar or unconventional, which is why it was chosen amongst other sorting algorithm because visualizing it would highlight its difference compared to other sorting algorithm and would also help grasp its behaviour and approach.

The idea behind the algorithm is to have the elements to be sorted as a type which the algorithm transforms into ordered chain. The algorithm only works for positive numbers. The representation of the unsorted entries had to be different as well to match the unconventionality of the algorithm

First step is to implement the algorithm in CHR, we find that only two rules are responsible for the whole operation of the sorting.

```
sort(X) <=> 0 <<< X.
A <<< B \ A <<< C <=> A<B, B=<C | B <<< C.
```

From the above two rules, we can try to understand the algorithm. The `sort(Number)` constraint takes one argument which is the number itself, thus each sort constraint represents one element and therefore when the user enters the input, a sequence of `sort/1` would be provided. Each of these constraints would fire the first rule in the above two rules, this rule would create a constraint `<<</2` which represents a chain. This chain created by the first rule is between the argument, which is one of the elements to be sorted, and 0. Granted that the numbers provided are positive numbers the chain start from 0 to the other. This shows that every element is connected to the 0 at first. This is something to be considered while thinking of the structure depicting the implementation. The second rule is responsible for the merging of the constraints, whenever there are two chains in the constraint store with common smallest element, they are merged into a chain from the smaller to the larger, while keeping the constraint that holds the smaller element of the result as the larger. Example, having:

```
1 <<< 2 , 1 <<< 3
```

This would result into the addition of this constraint 2 <<< 3 to the constraint store alongside 1 <<< 2 and the now redundant 1 <<< 3 is removed.

As evident from the explanation of the code, this algorithm is complex and as already stated unconventional, so it could become cumbersome to trace it, debug it or, for a student new to CHR, even understand it.

After the implementation, the next step is to think how this algorithm could be visualized or depicted depending on its behaviour. This was the toughest step in this implementation, thinking of the structure and layout to be created for the visualization.

Considering the chains and arcs connecting elements is somehow similar to a graph, the structure chosen was a graph but the nodes had to be stationed in a specific manner.

The implementation showed that the execution results in every element connection to the 0 at some point and the possibility of connecting to any other element. The ability to have the option of creating a straight line from any node to another restricted the possibilities of layout to a few ideas.

The one chosen was to have the node laid in a circular layout that is to arrange the entries in a circular manner while adding the 0 element in the

middle. Thus, this makes it possible to connect an arrow between any two numbers and it is possible to connect all the numbers with the 0 as well.

This circular layout needs the number of elements as an input due to its importance in creating a symmetrical layout, as the number of elements would decide the angle difference to the origin between two consecutive elements. For this reason the constraints `num/1,info/5` were added. The first must be appended to the input as it is used to initialize the screen.

The constraint `num(Num)` has one argument which is the number of elements, this used to calculate the angle between the entries. It is also used so the screen would be initialized just once.

The constraint `info(OldAngle,NewAngle,CX,CY,Distance)` just holds some information about the animation, it holds constants of the animation, $OldAngle$ is the angle in which is located the current element, $NewAngle$ is the angle of the next element, $CentreX$ and $CentreY$ are the coordinates of the centre where the 0 is drawn and finally $Distance$ is the radius of the circle having the 0 as its centre.

The input should have the constraint `num/1`followed by a sequence of constraints `sort/1` which represent the unsorted elements. Then the input would first fire this rule

```
num(N) <=>  Angle is ((360/N) * (3.14 / 180)),
            Distance is N + 75,
            initialize_screen(black),
            info(0,Angle,300,300,Distance),
            variable(0,@zero,300,300).
```

When the body of the rule is applied, first the $Angle$ is calculated by dividing 360 over the number of elements, then the radius is set, afterwards the screen is initialized by calling the predicate

```
initialize_screen(Color) :-
   new(@window, window('Sorting',size(600,600))),
   send(@window, open),
   new(@zero,text(0)),
   send(@zero, font, font(times, bold, 14)),
   send(@zero,colour,colour(Color)),
   send(@window,display,@zero,point(300,300)).
```

The predicate creates the window, creates the 0, colours it and then adds it to the screen.

After the screen is initialized the constraints `info/5,variable/4` are added to the constraint store.

The constraint `variable(Value,Id,X,Y)` holds information about each element, it holds the value and associates it to a certain identification and the location.

Continuing on the rules of the input, on account of always having the constraint `info/5` ,each `sort` constraint would fire this rule

```
sort(N), info(Angle1,Angle2,X,Y,D) <=>
                    NewAngle is (Angle2-Angle1)+ Angle2,
                    Xco is X + (D*sin(Angle1)),
                    Yco is Y + (D*cos(Angle1)),
                    info(Angle2,NewAngle,X,Y,D),
                    variable(N,Id,Xco,Yco),
                    draw_number(N,Id,Xco,Yco),
                    connect_numbers(300,300,Xco,Yco),
                    0 <<< N.
```

This rule means the addition of a new element N to the chain, uses the current angle and the angle difference to value the next angle, then it uses the angle, the radius and the position of the centre to calculate the element's x and y coordinates. Afterwards a new `info/5` is added to the constraint store, along with a `variable/4` for the new element. After the `variable/4` has been added, the predicate responsible for drawing the element at its given location is called.

```
draw_number(N,Id,X,Y):-
   new(Id,text(N)),
   send(Id, font, font(times, bold, 14)),
   send(@window, display, Id, point(X,Y)),
   send(@window,flush).
```

The predicate just creates the node, and its identification is assigned to the unbound variable Id in the constraint `variable/4` of the new element, adds it to the window in the specified location, and finally a repaint is called to make the newly drawn number appear.

Following the calling of the predict that draws the number, the constraint that connects two numbers is added to the constraint store to connect in the animation the newly drawn number with the 0. This is achieved after the below rule is fired

```
connect_numbers(X1,Y1,X2,Y2) <=> sleep(1),
                                 line2(X1,Y1,X2,Y2,Id).
```

The constraint is simplified to a prolog predicate that adds a delay of one second and another CHR constraint `line2/5` which has 5 arguments: the coordinates of the start and end points of the line and the identification set and associated with that line.

This consequently fires the below rule, which is a propagation rule that serves the purpose of calling the predicate `set_margin/5`.

```
line2(X1,Y1,X2,Y2,Id) ==> set_margin(X1,Y1,X2,Y2,Id).
```

The predicate `set_margin/5` just defines an offset to the start and end of the line to shape a distance between the actual number and the line while considering the arrow as well.

```
set_margin(X1,Y1,X2,Y2,Id):-
              (
              (X2 =< X1, Y2 >= Y1, OfX1 is X1-5,
              OfY1 is Y1+5, OfX2 is X2+5, OfY2 is Y2-5);
              (X2 < X1, Y2 < Y1, OfX1 is X1-5,
              OfY1 is Y1-5, OfX2 is X2+5, OfY2 is Y2+5);
              (X2 > X1, Y2 < Y1, OfX1 is X1+5,
              OfY1 is Y1-5, OfX2 is X2-5, OfY2 is Y2+5);
              (X2 > X1, Y2 > Y1, OfX1 is X1+5,
              OfY1 is Y1+5, OfX2 is X2-5, OfY2 is Y2-5)
              ),
              new(Id,line(OfX1,OfY1,OfX2,OfY2,second)),
              send(@window,display,Id),send(@window,flush).
```

Finally, after the 0 is connected with the newly created number the constraint `>>>/2` which represent the chain is created between the new element and the 0.

AS explained above in the original CHR implementation, the rule below is responsible for the merging of the constraints. This merging sorts the array after the below rule is no longer applicable to any two constraints. For the purpose of the visualization the implementation was changed after embedding the code that would create the animation.

```
variable(A,Id,X,Y),
variable(B,Id1,X1,Y1),
variable(C,Id2,X2,Y2),
line2(X,Y,X1,Y1,Id3),
line2(X,Y,X2,Y2,Id4),
A <<< B \ A <<< C <=>
           color(Id,red), color(Id1,red),
           color(Id2,red), color(Id3,red), color(Id4,red),
           A < B, B =< C | send(@window,flush), sleep(1),
           color(Id,black), color(Id1,black),
           color(Id2,black), color(Id3,black),
           color(Id4,black), remove_line(Id4),
           connect_numbers(X1,Y1,X2,Y2),
           B <<< C.
```

The head included five additional constraints, a `variable` constraint for each element which would be useful so we can have their identifications at hand, thus making it three `variable` constraints, also two lines constraint for each line representing the chain between each pair. The execution of the body of the rule yields to colouring each node and each chain to red so that the elements that are checked in the guard whether to participate in the merging are highlighted before hand. Afterwards if the guard matched then a delay of one second is added and the window is repainted. This is followed by the lines and variables coloured back to black and the line corresponding to the removed chain being removed as well by adding the constraint `remove_line/1`.
After the removal, `connect_numbers/4` is added to the constraint store to connect the two merged elements, then the chain itself between the merged elements is added. Removing the line is accomplished after the constraint added to the constraint store fires the below rule, which matches the constraint that ask for line removal, with the constraint that represent the corresponding line and simplify the head to `free/1` which frees the identification of the line and therefore remove it from the window.

```
remove_line(Id), line2(_,_,_,_,Id) <=> free(Id).
```

This algorithm would be considered to belong to the second category as a graph based represented was used to visualize it.

The figures below show snapshots of animation to the query:

```
num(10),sort(16), sort(12), sort(11), sort(14),
sort(13),sort(6), sort(2), sort(1), sort(4), sort(3).
```

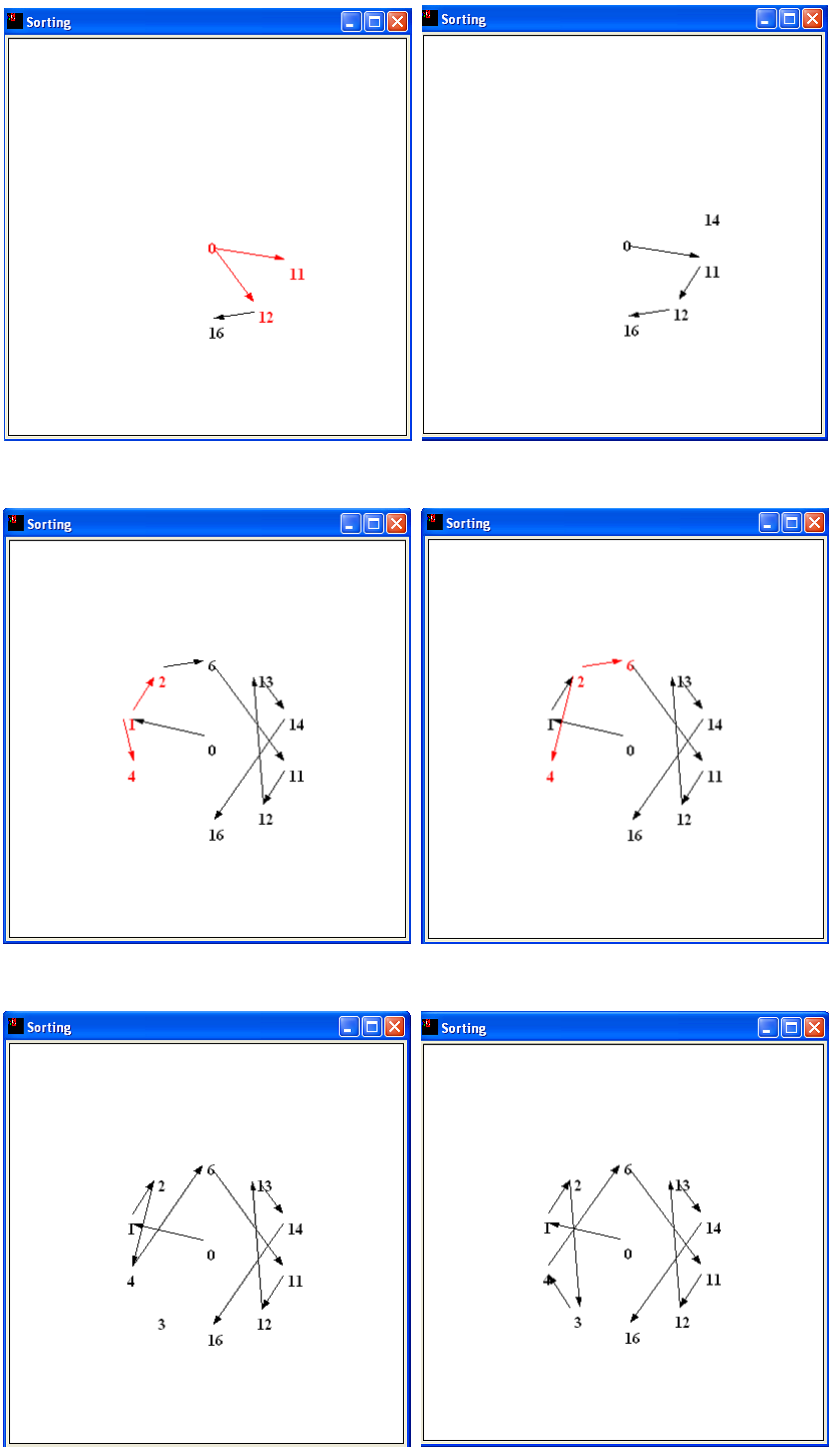


Figure 4.4: Ordered Merge Sorting

4.2.3 Exchange Sort

After representing an unfamiliar sorting algorithm, this algorithm was chosen for the complete opposite, as exchange sort resembles traditional sorting algorithm. This is obvious from its name, since exchange sort was the second of three cluster created by Baecker in his work on Sorting algorithm visualization [1] Exchange sort is a sorting algorithm that exchanges neighbouring elements if they are in the wrong order whose CHR implementation only consists of one rule:

```
ar(I,V), ar(J,W) <=> I>J, V<W | ar(I,W), ar(J,V).
```

In this algorithm, an array is a sequence of constraints `ar(Index,Value)`. For the purposes of the visualization, the implementation was changed to add a few predicates and rules that help the process of visualizing and showing the animation.

Since this algorithm as mentioned resembles conventional sorting algorithms similar to the ones visualized in the Sorting out Sorting video, thus the idea for the representation was to have the algorithm represented as the Sorting out Sorting video. On a bar chart with each element represented by a bar with the bars interchanged according to the algorithm itself

First, to start the animation the user has to enter an input that starts with the constraint `max(MaxValue)` which specifies the largest element in the list of numbers, afterwards the array is provided in the form of a sequence of constraints `ar(Index,Value)`.

This input would fire many rules. First, this one

```
max(X) <=> init_screen(X).
```

Thus the screen is initialized after the body of the rule is applied. The predicate uses the maximum value in the initialization of the bar chart, this is done after the picture has been created and the bar chart as well. The initialized bar chart is then added to the screen with the y axis ranging from 0 to a value slightly larger than the largest element of the sequence.

```
init_screen(X):-
        new(@window, picture('Exchange Sort',size(600,600))),
        send(@window, open),
        Top is Max + 5,
        new(@chart,bar_chart(vertical, 0, Top, 500, 20)),
        send(@window,display,@chart).
```

The rest of the input is the sequence of `ar(Index,Value)` constraints. Each one of these constraints represents an element of an array, where the first variable represent the index within the array and the second variable represents the value itself of that index. Thus making the sequence of these constraint represent the array.
There are two constraints `ar(Index,Value)`and `array(Index,Value,ID)`, since one is used merely as an intermediate form that is used just once for each element to draw its bar and give it the identification of the bar. Then it is transformed to the other form, which is the other constraint that is responsible for the rest of the execution. This is achieved by the rule below that is fired by each `ar/2` constraint in the input

```
ar(I,V) <=> draw_element(I,V,Id),
            send(@window,flush),
            array(I,V,Id).
```

As evident the constraint is simplified to the predicate

```
draw_element(I,V,Id):-
    sleep(1),
    send(@chart,append,new(Id,bar(I,V,blue))).
```

which is in charge of drawing the element in a form of a bar and then adding it to the bar chart after a delay of one second, the picture is repainted to make sure the newly drawn element appears and finally the other form of the element is created after the identification of the bar was added as an argument. The identification was appended as an argument to be able to manipulate any bar corresponding to certain constraint.

In a sorted array, for each pair `ar(I,V),ar(J,W)` with I $>$ J it should hold that V $\geq$ W.

The main rule that is responsible for the sorting operation itself and animation is fired whenever two `array` constraints are in the constraint store while one is having a smaller index with a larger value than the other which would mean they are in the wrong order. Thus this checked for if it is satisfied then they are in the wrong order and the values are exchanged.

```
array(I,V,Id), array(J,W,Id1) <=>
                          I > J, V < W | sleep(1),
                          send(Id,value,W),
                          send(Id1,value,V),
```

```
send(Id,colour,colour(yellow)),
send(Id1,colour,colour(yellow)),
send(@window,flush),
sleep(1),
send(Id,colour,colour(blue)),
send(Id1,colour,colour(blue)),
send(@window,flush),
array(J,V,Id1), array(I,W,Id).
```

As explained, the above rule is fired when two `array` constraints are in the constraint store, first the guard checks if one has a higher index while having a lower value to test the applicability of the rule. If the guard matched, it means that the elements are in the wrong order and have to be exchanged therefore the body is applied, this is shown in the visualization first as `send(Id,value,W), send(Id1,value,V)` exchanges the two corresponding bars' values while the bars are coloured in yellow to highlight the exchange. This appears on the screen after `send(@window,flush)` repaints the picture. Then there is a delay of 1 second, created by the Prolog predicate `sleep/1` which stops the execution for the specified amount of time, for the user to have sufficient time to absorb the change and for the algorithm to have smooth motion in the animation. After the delay, both of the bars are coloured back to their original blue after the bars have been exchanged, as usual this appears after the repaint. Finally the values of the elements inside the constraints are exchanged.

As shown, every time the above rule is applied, it corrects at least one ordering. This is performed for the rest of the elements till the rule is no longer applicable which would mean that the elements had become sorted, on account that it is not applicable, it means that every element's all larger valued elements have higher indices, thus a sorted list. Program is confluent for queries with known numbers, but not in general

This implementation belonged to the third category, the algorithm that could be represented by bar graphs.

The below figures show the snapshots of the visualization for the query:

```
max(9),ar(0,1), ar(1,7), ar(2,5), ar(3,9), ar(4,2).
```

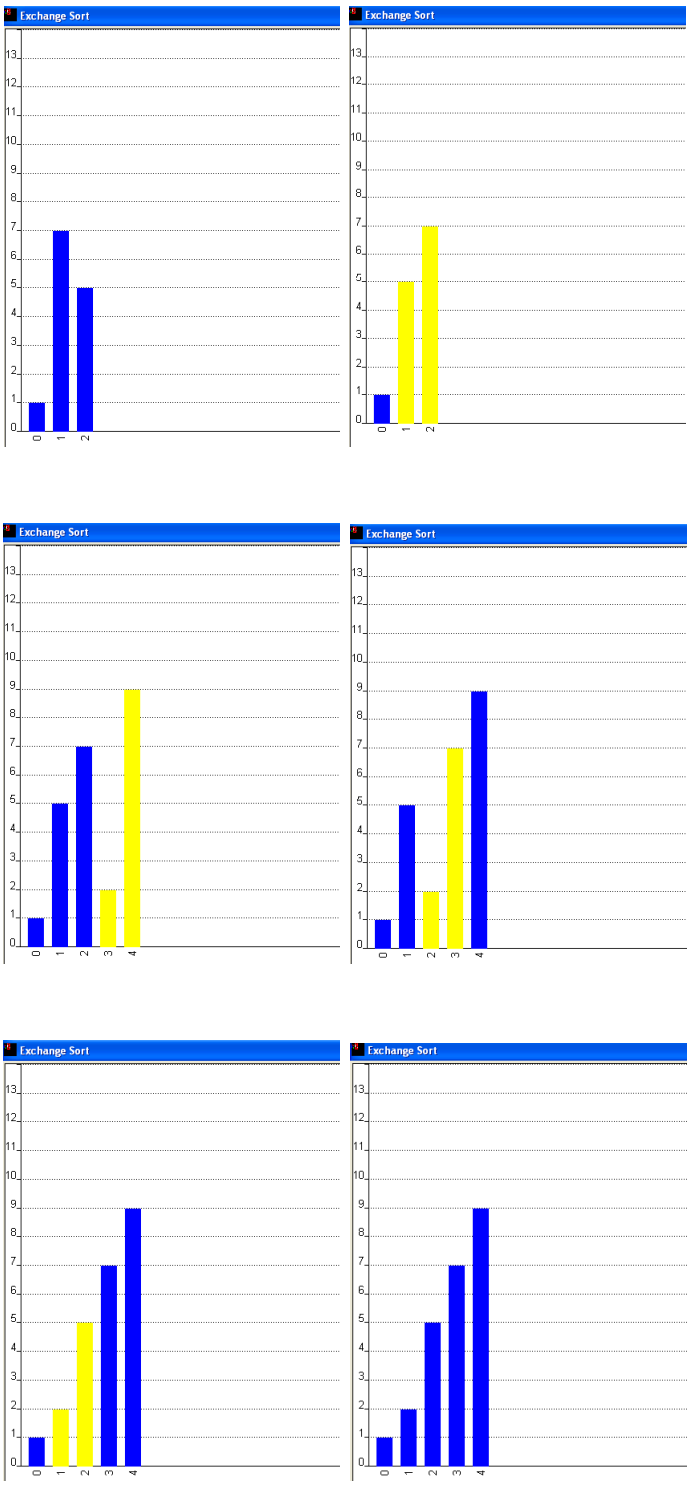


Figure 4.5: Exchange Sort

4.2.4 Depth First Search

Depth First Search(DFS) is of the most important, well known and immensely used tree traversal searching algorithm.

Moreover, in 2002 , Abdennadher et al. created a constraint library for Java, one of its constituents was a a visualization tool for the propagation and simplification of constraint [16]. This could be considered as one of the first visualizations for CHR, one of the future work targets of that work was to create a visualization for search trees.

Thus the implementation of depth first search (DFS) in CHR had to be visualized and shown.

The general idea of Depth first search in a tree is to start at the root, then from there the algorithm travels as deep as possible from neighbour to neighbour before backtracking.

As we mentioned in Chapter 2, most of the algorithm visualization systems are in Java, making the available DFS animations mostly the ones programmed imperatively, which differs significantly from the declarative implementation.

The animation shows how different the CHR version of the DFS really is, since as evident in the animation what happens is that the search is happening while building the tree more or less

As always the first step was to implement the algorithm in CHR, we would find that the implementation consists of three rules.

```
dfs(leaf(Val),X) <=> X == Val.
dfs(node(Val,_,_),Val) <=> true.
dfs(node(Val,L,R),X) <=>
X \== Val | (dfsearch(L,X) ; dfsearch(R,X)).
```

In the above implementation, Val represents the value of the node, L and R are trees representing the left subtree and the right subtree respectively. A tree could be one of two either a node with right and left subtrees `node(Val,L,R)`, or a `leaf(Val)`. To perform a depth first search for a target X in a tree The input should be of that sort: `dfs(T,X)`.

Each rule in the implementation covers one of the cases that could occur. The first case if the current node is a leaf, then the output would be false if the value is not the target and true if it is. The second case is

that the current tree is a node with the value of the target hen the output is true. The third case is to have the current tree as a node whose value is different from the target, at that point the right tree and the left are searched.

The phase of the selection of the suitable representation for the algorithm was virtually non-existent, since this is a tree traversal algorithm, it could not be visualized as anything but a tree.

The work it took to visualize this was relatively easy, as the combination of XPCE's rich powerful built-in library classes with some of the advantages of logic programming, made it easy to add just a few lines of code for the visualization.

First there had to be something that creates the scene just once, so the `dfs/2` constraint entered as an input is used as an auxiliary constraint that just sets the scene and then is transformed to anther form. This is shown in the three rules below

```
dfs(leaf(Val),X) <=> ((Val = Id, draw_tree(Val,X,Id,green));
                            draw_tree(Val,X,Id,red)).
dfs(node(Val,_,_),Val) <=>draw_tree(Val,Val,Id,green).
dfs(node(Val,L,R),X)<=> X \== Val |
                            draw_tree(Val,X,Id,red),
                           (dfsearch(L,X,Id);dfsearch(R,X,Id)).
```

The three cases are covered as well, the first case, the value of the leaf is checked if it is equal then the screen is initialized and the tree is drawn with a green coloured root as a sign of success resulting in stopping the algorithm and animation, if it is not equal then it means that the target does not exist and therefore the tree is drawn with a red root. The second case is similar to the success story of the first, since the value of the node is equal to the target then the tree is drawn with a green coloured root as a sign of success. The third case, which should be the common case, the screen is initialized, the root of the tree is drawn and the execution continues after the transformation to the other form that hold an additional argument which is the identification of the tree. This is appended to be able to draw the child of the tree.

As evident, the predicate `draw_tree/4` is called in the three auxiliary rules to initialize the screen, it takes four arguments used for the initialization.

```
draw_tree(Root,Target,Id,Color) :-
```

```
    new(@d, window('Depth First Search',size(500,500))),
    send(@d, open),
    send(@d, display,
        new(TargetText, text('Target', center, large)),
        point(400,375)),
    send(TargetText,colour,colour(blue)),
    send(@d, display,new(TargetBox, box(50, 50)),
        point(400,400)),
  send(@d, display,
      new(TargetNumber, text(Target, center, large)),
      point(425,425)),
    send(TargetNumber,colour,colour(blue)),
    sleep(1),
    new(@t,text(Root)),
    send(@t, font, font(times, bold, 14)),
    send(@t,colour,colour(Color)),
    new(T, tree(new(@r, node(@t)))),
    Id = @r,
    send(@d,display,T),
    send(@d,flush).
```

First the window is created, then the word 'target' is drawn in the bottom right corner of the screen using the colour blue , right under it a square is drawn, afterwards the search target is created coloured in blue and put inside the square. A 1 second delay is added before creating the root in the specified colour and adding it to the tree and the screen. The argument Id was unbound when calling the predicate and it was set to the root since this would become the parent to its subtrees.

The transformation to the other form `dfs/3` results in the continuation of the execution and building the rest of the tree.

```
dfsearch(leaf(Val),X,Id) <=>
                ((X == Val,sleep(2),
                draw_node(Val,Id,_,green));
                (sleep(2),draw_node(Val,Id,_,red),false)).

dfsearch(node(Val,_,_),Val,Id) <=>
                        sleep(2),
                        draw_node(Val,Id,Id2,green).

dfsearch(node(Val,L,R),X,Id) <=>
                           X \== Val|
```

```
                    sleep(2),
                    draw_node(Val,Id,Id2,red),
                   (dfsearch(L,X,Id2);
                     dfsearch(R,X,Id2)).
```

The same three cases exist with a few animation actions appended to the implementation. The first case occurs if the algorithms reaches a leaf, the value is checked with the target in case of equality, then the node is drawn in green, subsequently the animation is stopped and the algorithms yields true. In case of inequality then the node is drawn and returns false.

The second case, again is the same as the success story of the first, since the value of the node is the same as the target, the node is drawn in green, subsequently the animation is stopped and the algorithms yields true.

The third case occurs when the subtrees have to be searched as well, thus `draw_node/4` is called to draw the current node and a recursive call is made on the right and left trees.

The predicate `draw_node/4` is in charge of drawing the new nodes

```
draw_node(X,Id,Id2,Color):-
                  new(Q,text(X)),
                  send(Q, font, font(times, bold, 14)),
                  send(Q,colour,colour(Color)),
                  new(Id2, node(Q)),
                  send(Id, son,Id2),
                  send(@d,flush).
```

The predicate takes four arguments: the value, the identification of the parent for which the child will be drawn, an unbound variable $Id2$ which will be bound to the identification of the newly drawn child node and finally the colour used for drawing the node. Similarly to the creation of the root, the text is drawn in the specified colour, the node is created with the text, the difference here is that `send(Id, son,Id2)` is used to connect the node to its parent.

This algorithm belonged to the second category, since it is a graph based algorithm.
Below figure 4.6 shows a running of the algorithm for this query:

```
dfs(node(0, node(1,node(2,node(5,leaf(8),leaf(9)),leaf(7)),
      node(10,leaf(14),leaf(3))),
    node(6,node(16,node(21,leaf(50),leaf(72)),leaf(31)),
      node(44,node(12,leaf(35),leaf(33)),leaf(4)))), 4).
```

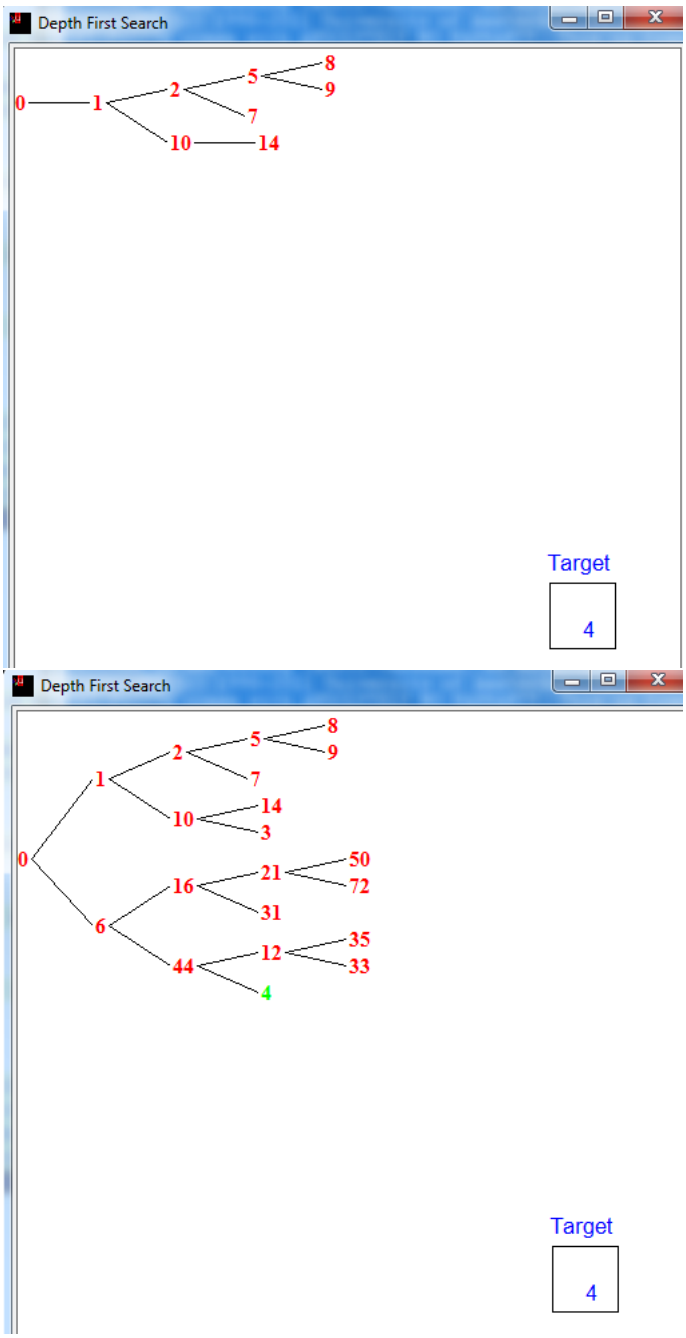


Figure 4.6: Depth First Search

4.2.5 Ballistic Trajectory Simulation

Ballistics is a branch in mechanics that studies projectiles, their dynamics, types and flight statistics. Ballistic trajectory is the trajectory or the path created by an object in space with no regards to any other resistive force

while being under the influence of only gravity. The trajectory depends on a couple of variables, it is decided by the initial height of the surface the object is stationed at, initial velocity or the force applied on the object, the direction and the angle of the force applied and finally the gravitational force of the environment. These inputs would determine the exact path, the exact coordinates of the object at a certain point in time, the maximum distance travelled, maximum height reached and the total time taken to reach the ground. The formulae needed to calculate them are:

The maximum distance travelled by the object is calculated using this formula :

$$D = v_0 t_d \cos\theta \tag{4.1}$$

The maximum height reached :

$$H = \frac{v_0^2 \sin^2\theta}{2g} \tag{4.2}$$

The total time taken to reach the ground :

$$T = \frac{2v_0 \sin\theta}{g} \tag{4.3}$$

To get the x-coordinate at a certain point in time :

$$x = v_0 t \cos\theta \tag{4.4}$$

To get the y-coordinate at a certain point in time :

$$y = v_0 t \sin -\frac{1}{2} g t^2 \tag{4.5}$$

Angry Birds is one of the most popular and widely played video games around the world, the game was developed mostly relying on the concept of ballistic trajectory. After accompanying that with some features, great graphics and animation, the game was one of the best seller applications on all mobile platforms.

These facts were factors that helped inspire this implementation and animation afterwards. Since CHR is a declarative rule based programming language, thus defining the rules mentioned above that would enact the ballistic trajectory would be smooth and easy.
This example was implemented from scratch, the CHR code was implemented and then the XPCE code was added in the fundamental rules to produce the animation. The start of the CHR code was: first the rule that would start the simulation was created, the constraint `launch/3` was used for the start the constraint has an arity three. The three arguments are the velocity, the firing angle and the gravitational force of the environment, as these are the inputs needed for the calculation as pointed out earlier.

```
launch(V,Angle,G) <=> get_initX(InitX),
                      get_initY(InitY),
                      A is Angle * (3.14 / 180),
                      calculate_time_to_gorund(V,A,TT,G),
                      calculate_max_height(V,A,MaxH,G),
                      calculate_max_distance(V,A,TT,MaxD),
                      init_screen(InitX,InitY,MaxD,MaxH),
                      xcord(X,V,A,TT,InitX,T1,G),
                      ycord(Y,V,A,TT,InitY,T2,G),
                      time(TT).
```

The rule above is responsible for the start of the simulation, it starts by setting the Initial coordinates of the objects, with the help of the two predicates

```
get_initX(InitX), get_initY(InitY)
```

Afterwards, the angle is converted to radian and passed to the three auxiliary predicates below to get the trajectory's characteristics.

```
calculate_time_to_gorund(V,A,TT,G),
calculate_max_height(V,A,MaxH,G),
calculate_max_distance(V,A,TT,MaxD)
```

The first predicate uses the variables representing Velocity, Angle and Gravity to calculate the total time it takes the object to reach the ground and set it to the unbound variable TT using equation 4.3. This predicate also helps with the termination of the animation, as the `time(T)` constraint which is used in drawing the object and the update of its location, is started with the total time and decrease with each draw until it reaches 0.

The unbound variable $MaxH$ is matched to the calculated maximum vertical height reached by the object in the second predicate which implements equation 4.2 .

The third predicate uses the total time TT as well as the other inputs in equation 4.1 to match the unbound variable $MaxD$ to the maximum horizontal distance travelled by the object.

The acquisition of the total time, maximum horizontal distance and the maximum vertical height reached paves the way to the initialization of the screen.

```
init_screen(InitX,InitY,MaxD,MaxH)
```

The above predicate initializes the screen, it provides the start point's coordinates, the maximum distance and height. First the picture and the dialog are created, then the object is created in a form of a circle and stationed at the start point. Subsequently, the constraint `obstacle(X,Y)`, that holds the coordinates of the obstacle, was added to the constraint store, which fired the rule below that creates the obstacle. This rule is simplified to the XPCE code that draws the obstacle and add it to the picture in the specified location.

```
obstacle(X,Y)==>new(@obstacle,bitmap('Obstacle.xpm')),
                send(@window,display,@obstacle,point(X,Y)).
```

Afterwards, an x-axis and a y-axis are drawn, the y values range from 0 till the maximum height and the x axis ranges from the 0 till the maximum horizontal distance the object would end at in case it did not hit the obstacle.

```
send(@window, display,
             plot_axis(x, 0, MaxD, @default,
                         600, point(InitX, InitY + 15))),
send(@window, display,
             plot_axis(y, 0, MaxH, @default,
                         500, point(InitX, InitY + 15))),
```

These axes could serve as an educational tool as well due to the fact that the user would have an idea about the different exact distances in numbers and trajectories of an object after the effect of different forces at various angles.

The basis of the update of the x and y coordinates throughout the animation is three constraints

```
ycord(YCordValue,InitVelocity,FiringAngle,
        TotalTime,InitY,Time,Gravity),
xcord(XCordValue,InitVelocity,FiringAngle,
        TotalTime,InitX,Time,Gravity),
time(T)
```

These are the last constraints in the body of the rule above. The `ycord` and the `xcord` are constraints that hold the current y and x coordinates respectively with the info of the trajectory as well. The `time` constraint has one argument which is the time remaining for the object to reach the ground, this constraint is simplified before each draw and then a new time constraint with the new remaining time is put in the constraint store after the draw, this happens till it reaches 0. With the three constraints activated, the simulation starts, consequently they would fire the rules responsible for the beginning of the movement of the object.

```
xcord(X,V,A,TT,InitX,T1,G),
  ycord(Y,V,A,TT,InitY,T2,G)\ time(T) <=>
                       var(X),var(Y)|
                       CT is TT - T, T1 = T, T2 = T,
                       X is InitX + (V * CT * cos(A)),
                       Y is InitY -((V * CT * sin(A)) -
                                    (G * CT^2)/2).
```

The body of this rule is applied if the guard is satisfied. The guards checks whether, at a certain time remaining T, the variables X and Y, which represents the values of the x and y coordinates respectively, are bound or not. In case they are not bound then the body of the rule applies, it calculates the time passed or the current time(CT) by subtracting the time remaining(T) from the total time. Afterwards the values that should be assigned to the xcord and ycord at the given time is calculated, using equations 4.4 and 4.5 respectively, and matched to the unbound variables X and Y.

Once these two variable become bound, one of the two rules below is fired, depending on the location of the object

```
obstacle(X,Y)\
 xcord(X1,V,A,TT,InitX,T,G),
  ycord(Y1,V,A,TT,InitY,T,G)
   <=> number(X1), number(Y1),
    (X1 < X - 12; X1 > X + 12;
     Y1 < Y - 20; Y1 > Y + 20)| draw(X1,Y1,T),
                               xcord(X2,V,A,TT,InitX,T1,G),
                               ycord(Y2,V,A,TT,InitY,T2,G).
```

The above simpagation rule is responsible for displaying the movement of the object along its trajectory. As long as the object is not within

touching distance of the obstacle, the object is drawn with the constraint `draw(XCordValue,YCordValue,Time)` which would be discussed later and new `xcord/7` and `ycord/7` with non bound value are put in the constraint store waiting for the time constraint to fire the rule that bounds the values. This rule was one of the things that made use of the syntax of CHR since a simple simpagation rule would do the trick of identifying the collision and then removing the old coordinates while leaving the obstacle intact as well

```
xcord(X1,V,A,TT,InitX,T,G),
ycord(Y1,V,A,TT,InitY,T,G),
obstacle(X,Y) <=> number(X1), number(Y1),
                  X1 > X - 12, X1 < X + 12,
                  Y1 > Y - 20, Y1 < Y + 20 |
                  draw(X,Y,T), killed.
```

This simplification rule handles the collision detection, the guard checks if the difference between the object's and the obstacle's x-coordinates lower than 12 which is the width of the obstacle and if the difference of y-coordinates less than 20, if these requirements are satisfied then the object has became within touching distance of the obstacle. Once the object is within touching distance of the obstacle this rule is fired, the constraint `draw`draws the object at the final location and then the constraint killed is fired which executes the consequences of the collision.

The constraint `draw` deals with the drawing of the object and is also responsible for the generation of new time constraint. Each time the rule below is fired, the object is drawn on the picture in the point specified in the argument of the constraint. Subsequently the picture is repainted to display the new location of the object and then the time remaining is decreased and the execution waits for 0.05 seconds and a new time constraint with the new remaining time is put in the constraint store

```
draw(X,Y,T) <=> new(@ball,bitmap('Object.xpm')),
                send(@window, display,@ball, point(X,Y)),
                send(@window,flush),
                T1 is T -0.2 ,sleep(0.05), time(T1).
```

For the purpose of this visualization, there were numerous decisions to take during the implementation, there was the choice of what to set as the travelling object, the selection of the obstacle and finally the actions that would occur in the animation after the collision

Two options were implemented, the first had the object as a ball and the obstacle as a circle which disappears after colliding with the ball as if the obstacle was a coin and the ball collected it. The second is the one represented in this work, It has the object as a football and the obstacle a goal, having the target to score a goal. The collision detection results in stopping the simulation and appearing the label goal in the footer.

This implementation belong to the final category where the elements had to be depicted using special bitmap images. The program is confluent for queries with known numbers, as the order of the rules applied will not affect the final output. This example did not just show how efficient CHR is when it comes to rule based algorithms but it also went on to visualize it and show that the language could be used in programming games and light applications as well

A winning scenario, where the goal is hit , is shown in figure 4.7

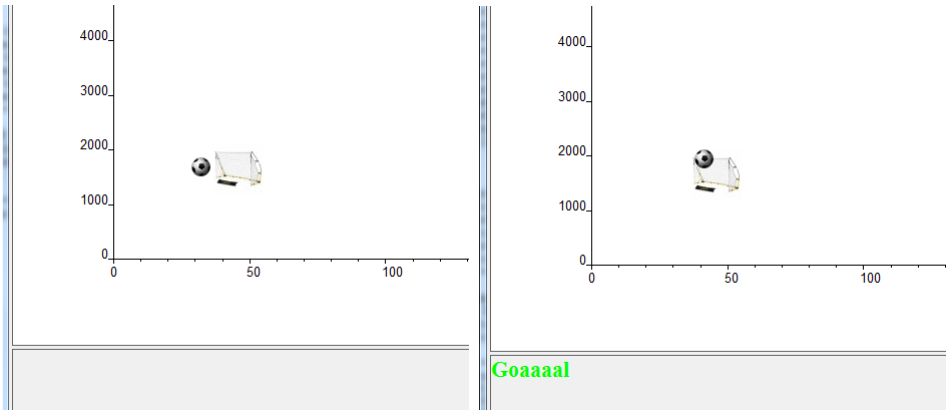


Figure 4.7: Winning trajectory

A losing scenario where the ball does not go into the goal and completes its path till it reaches the ground is shown in figure 4.8 Query: `launch(70,60,9.8).`

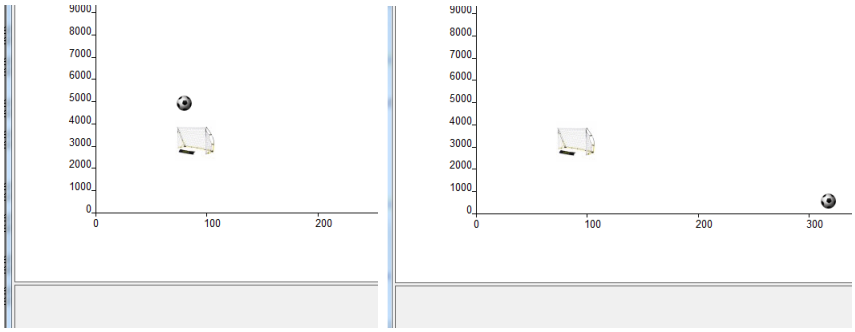


Figure 4.8: Losing trajectory

4.3 Related Work

The Algoviz wiki which we discussed earlier showed that there are still no genuine accountable algorithm visualization tool created using CHR or for CHR programs. The research performed in this area showed that only few efforts were made in this topic, two efforts [17] [16] stood out and were the only convincing trials, however they were basis, the representation did not really enact the behaviour of the algorithm and the tool was created using JCHR which as we discussed in section 3.1 has decreased in popularity and support over the last few years.

Thus there are no significant tool to be compared with this one except for the tool that was developed at the same time as this work [18]. Due to the simultaneity of the development of that work and this one, a clear grasp of the visualizations and the functionalities of the tool itself was not a possibility.

However an idea about the tool and its functionalities was given, the tool takes a Prolog source code and then performs a source to source transformation to be able to visualize the algorithm using Java. This work differs significantly from the aforementioned tool since it does not need source to source transformation as the visualization is done using XPCE and the XPCE code is embedded within the Prolog source code to create the animation.

The necessity to have source to source transformation to be able to create the visualization could prove ineffective. As the stability of the source to source transformation is responsible for a correct visualization and if the source to source transformation can not be applied on certain complicated algorithms , these algorithms will not be visualized.

Also, the effectiveness of using a transformed program as an educational tool would be less than using the program itself after appending the code responsible for the visualization and therefore instead of the ability to teach students through the visualization and the code, to highlight the execution and events, the source to source transformation tool would only help understanding the algorithm through the visualization.

Chapter 5

Conclusions

5.1 Conclusion

As a summary of this work's content, we came to the discovery that software visualization is an important broad field in computer science that has many applications, however it is not addressed as much as it should be. The research on Software visualization contrary to beliefs goes back before the 1981 algorithm animation 'Sorting out Sorting' video. Algorithm animation(AA) is a fundamental subtopic of Software visualization, its main application should be pedagogy, however AA systems are not used as much as they should be since programmers tend to focus more on graphical factors rather than addressing the pedagogical requirements and the needs of the students.

CHR is a high level uprising declarative programming language, however with the level of sophistication of the language, it becomes hard to understand the algorithms, their behaviour and how they operate and sometimes the textual tracing is not enough. Also it could be cumbersome to try to teach the unconventional algorithm to people new to the language while relying on textual explanation as well.

During this work, a research was conducted to uncover criteria and requirements already defined in this field. It was discovered that these criteria differentiate effective algorithm animations. This helps to create algorithm visualizations that could be utilized for educational purposes.

We clustered the algorithms implemented in CHR in terms of visualizations into four categories. Certain visual objects such as (Graphs and trees, grids, charts and Bitmaps) were assigned to each category which best suits the algorithm's data. Each of these categories had unique characteristics that define the algorithm's behaviour, structure and evolution of execution.

This involved settling on the possible events that could occur during the execution of the CHR program.

A set of steps was created as a guideline for the visualization of any algorithm implemented in CHR. These steps were applied on several algorithms from various clusters. XPCE was used as the graphical component for the visualizations. The XPCE code was embedded within the CHR rules, to show the various events.This was realised in five example algorithms that covered most of the clusters.

The aim of the work was thus realised by defining the framework for visualising any algorithm, and as proof of concept the application to five CHR programs was accomplished.

5.2 Future Work

This work has paved the way for future enhancements. The addition of functionalities in the already implemented algorithms could be addressed. The current occurring event and snippets of the source code responsible for the action could be displayed whilst the execution to highlight the CHR events. Moreover, the constraint store's content could be displayed in highlighted text alongside the animation and the source code. It should be contemplated how these possible additions would all fit in the animation.

Also, formalizing and generalizing the visualizations could be considered, while making use of the framework defined for the animation of an algorithm, this could be realised by allowing the user to add tags and annotations to customize the visualization.

Finally, a study could be made on learning and perception to investigate how people react to different structures and colours. The output of this study should be incorporated with this work to enhance the learning experience of the algorithms and their CHR execution.

Appendix A

Implementation JPL code

```
:- module(primes1, [prime/4]).
:- use_module(library(chr)).

:- chr_constraint  prime/4, upto/1,upto/5.

%creates the frame only once
upto(N) <=> N>1| R is sqrt(N),
                 initScreen(R,Grid,CS,F),
                 upto(N,Grid,CS,F).


upto(N,Grid,CS,F)<=> N>1| M is N-1, sleep(1),
                          addNumber(N,Grid,F),
                          upto(M,Grid,CS,F),
                          prime(N,Grid,CS,F).

prime(X,Table,Text,F) \ prime(Y,Table,Text,F)
                           <=> Y mod X =:= 0 |
                               atom_number(XStr,X),
                               sleep(1), remove(Y,Table,F).

numToString(N,S):-
    atom_number(A,N),
    atom_chars(N,L),
    atom_chars(S,L).

initScreen(R,Grid,CS,F):-
    R1 is floor(R),
    Width is R1*30,
```

```
    Height is R1*20 + 50,
    jpl_new( 'javax.swing.JFrame', ['Primes test'], F),
    jpl_call( F, getContentPane, [], CP),
    jpl_new('javax.swing.JTable', [R1,R1], Grid),
    jpl_new( 'javax.swing.JScrollPane', [Grid], SP),
    jpl_call( CP, add, [SP,'Center'], _),
    jpl_call( F, setLocation, [20,20], _),
    jpl_call( Grid, setSize, [Width,Height], _),
    jpl_call( F, setSize, [Width,Height], _),
    jpl_call( F, setVisible, [@(true)], _).



addNumber(X,Grid,F):-
    getRowAndCol(X,Grid,Row,Col),
    atom_number(XStr,X),
    jpl_call(Grid,setValueAt,[XStr,Row,Col],_),
    jpl_call(F, setVisible, [@(true)], _).

remove(Y,Grid,F):-
    getRowAndCol(Y,Grid,Row,Col),
    jpl_call(Grid,setValueAt,[' ',Row,Col],_),
    jpl_call(F, setVisible, [@(true)], _).

getRowAndCol(N,Grid,Row,Col):-
    jpl_call(Grid,getRowCount,[],NumOfRows),
    jpl_call(Grid,getColumnCount,[],NumOfColumns),
    Row is NumOfColumns - ceil(N/NumOfRows),
    Col is (NumOfColumns - (N mod NumOfColumns))mod NumOfColumns.
```

Bibliography

[1] R. Baecker, "Sorting out sorting : A case study of software visualization for teaching computer science," *Sort*, vol. 24, p. 369381, 1998.

[2] V. Karavirta, "Facilitating algorithm animation creation and adoption in education," masterslicentiate's thesis, Helsinki University of Technology, December 2007.

[3] C. Shaffer, M. Cooper, A. Alon, M. Akbar, M. Stewart, S. Ponce, and S. Edwards, "Algorithm visualization: The state of the field," *Trans. Comput. Educ.*, vol. 10, pp. 9:1–9:22, Aug. 2010.

[4] K. Knowlton, "A programmer's description of l6," *Commun. ACM*, vol. 9, no. 8, pp. 616–625, 1966.

[5] R. Baecker, "Two systems which produce animated representations of the execution of computer programs," *SIGCSE Bulletin*, vol. 7, pp. 158–167, 1975.

[6] J. Stasko, "Animating algorithms with xtango," *SIGACT News*, vol. 23, pp. 67–71, May 1992.

[7] G. Rößling, M. Schüer, and B. Freisleben, "The animal algorithm animation tool," *SIGCSE Bull.*, vol. 32, pp. 37–40, July 2000.

[8] W. Pierson and S. Rodger, "Web-based animation of data structures using jawaa," *SIGCSE Bull.*, vol. 30, pp. 267–271, Mar. 1998.

[9] T. Naps, J. Eagan, and L. Norton, "JhavÉ an environment to actively engage students in web-based algorithm visualizations," in *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, SIGCSE '00, (New York, NY, USA), pp. 109–113, ACM, 2000.

[10] G. Rößling and T. Naps, "A testbed for pedagogical requirements in algorithm visualizations," *SIGCSE Bull.*, vol. 34, pp. 96–100, June 2002.

[11] C. Shaffer, M. Cooper, and S. Edwards, "Algorithm visualization: a report on the state of the field," *SIGCSE Bull.*, vol. 39, pp. 150–154, Mar. 2007.

[12] T. Frühwirth, *Constraint handling rules*. Cambridge University Press, 2009.

[13] J. Wielemaker, P. Singleton, and F. Dushin, "Jpl." http://www.swi-prolog.org/packages/jpl/, June 2012.

[14] J. Wielemaker and A. Anjewierden, "Xpce." http://www.swi-prolog.org/packages/xpce/UserGuide/Contents.html, June 2012.

[15] P. Wuille, T. Schrijvers, and B. Demoen, "Cchr: the fastest chr implementation, in c," *Proceedings of the 4th Workshop on Constraint Handling Rules*, pp. 123–137, 2007.

[16] S. Abdennadher, E. Krmer, M. Saft, and M. Schmauss, "Jack: A java constraint kit," in *University of Kiel*, p. 2000, 2002.

[17] S. Abdennadher and M. Saft, "A visualization tool for constraint handling rules," in *In Proceedings of 11th Workshop on Logic Programming Environments, 1th*, 2001.

[18] 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.