# ALGORITHM ANIMATION IN PDF DOCUMENTS

**A PROJECT REPORT**

***Submitted by***

**T. MANTHOSH KUMAR**
**M. MUKUND**
**E. PRAKASH**

***in partial fulfillment for the award of the degree***

***of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

**SRI SIVASUBRAMANIYA NADAR COLLEGE OF ENGINEERING,**
**KALAVAKKAM – 603 110**

**ANNA UNIVERSITY :: CHENNAI 600 025**

**APRIL 2013**

# ANNA UNIVERSITY :: CHENNAI 600 025

## BONAFIDE CERTIFICATE

Certified that this project report **"Algorithm Animation in PDF Documents"** is the bonafide work of **"T. Manthosh Kumar (31509104063), M. Mukund (31509104065), E.Prakash (31509104076)"** who carried out the project work under my supervision.


**SIGNATURE**

**Dr. CHITRA BABU**

**HEAD OF THE DEPARTMENT**

Department of Computer Science and Engineering

Sri Sivasubramaniya Nadar College of Engineering
Kalavakkam - 603 110

**SIGNATURE**

**Dr. R. S. MILTON**

**SUPERVISOR**

**PROFESSOR**

Department of Computer Science and Engineering

Sri Sivasubramaniya Nadar College of Engineering
Kalavakkam - 603 110


Submitted for the Viva-voce examination held on _____

**INTERNAL EXAMINER**

Date:

Official Seal:

**EXTERNAL EXAMINER**

Date:

Official Seal:

# ACKNOWLEDGEMENTS

Foremost, we would like to express our deep and sincere gratitude to our supervisor *R. S. Milton*, Ph.D., Professor, Department of Computer Science and Engineering, for his continuous support to our project work, for his patience, motivation and enthusiasm. His guidance helped us throughout the course of the project. We wish to express our whole hearted thanks to *Dr. Chitra Babu*, Professor and Head of the Department for her encouragement and advice.

Besides, we would like to thank the project coordinator *Ms. S. Kavitha*, Assistant Professor and the B.E. Students Project Review Team for their encouragement, insightful comments and thought provoking questions.

It is a matter of pride and privilege for us to acknowledge our deep sense of gratitude to our Principal *Dr. S. Salivahanan*, our Dean *Dr. K. Kasturi* and our President *Mrs. Kala Vijaykumar* for their valuable support and encouragement.

We wish to avail this opportunity to express our deep sense of heartfelt gratitude to our family members for their inexhaustible love, care and continuous encouragement throughout career.

All glory to God for He makes the impossible become possible!

# ABSTRACT

The objective of the project is to be able to incorporate animated illustrations of algorithms in PDF documents for educational purpose. The PDF documents are generated using ConTEXt, a TEX macro that is tightly integrated with MetaPost, a powerful vector graphic programming language, and Lua, a modern scripting language. Users can implement, in Lua, algorithms manipulating data structures such as arrays, lists, trees, and graphs, and using the API provided, can animate the algorithms. The API handles the animation and the automatic layout of the data structure diagrams. The animation module also shows the pseudocode for the algorithm and highlights the currently executing statement.

Animation is a sequence of frames, with capabilities to play, pause, resume, navigate to next and previous frames, change framerate and skip certain frames when needed. The animation module is a modification of the t-animation in ConTEXt and implemented primarily using JavaScript. The data structure diagrams are drawn using MetaPost, and the layout using Lua. While an illustrative set of algorithms and data structures have been implemented, the main contribution of the work is to provide an API for the layout and animation which users can use for a variety of algorithms on these data structures. The documents generated can be viewed using the freely available Adobe Reader v6.0 or above.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface |
| CAD | Computer Aided Design |
| MAC | Macintosh |
| OCG | Optional Content Group |
| PDF | Portable Document Format |
| PGF | Portable Graphics Format |
| PRC | Product Representation Compact |
| SVG | Scalable Vector Graphics |
| TIKZ | Ti*k*Z ist kein Zeichenprogramm |
| 3D | Three Dimensional Space |

# CHAPTER 1
# INTRODUCTION

## 1.1 ANIMATED DIAGRAMS

Pictorial representation of information plays a critical role in the understanding of concepts, processes, and systems. Effective presentations and well written books use diagrams at appropriate places to clarify thought. When a process evolves dynamically with time, we use a sequence of diagrams to help one understand it. While printed documents allow only for static diagrams, electronic documents lend flexibility to diagrams. A sequence of frames can be used in a diagram to show the successive stages of a process, or parts of diagrams can be hidden or revealed to facilitate understanding. We will refer to them as animated diagrams, or simply *animations*. Specifically, we consider algorithms and the computational processes they engender. "Seeing" the computational process evolve is crucial to understand the properties of the algorithm design and show its correctness.

We intend to incorporate animated diagrams in PDF documents since PDF is a standard format. We have also decided to take the route of TEX and its macro package ConTEXt to generate the PDF documents. This boils down to be able to generate animated diagrams using TEX / ConTEXt. Quality drawing takes a lot of iterations before we arrive at a final acceptable result. It is better done interactively than programmatically. However, if the diagrams have a regular structure, they are amenable for programmatic construction.

## 1.2 ANIMATION IN PDF

Animation in PDF is brought about by Layers have been part of the PDF specifications since version 1.5. They are formally known as *Optional Content Groups (OCG)* and are in the PDF specification described as:

> "Optional content (PDF 1.5) refers to sections of content in a PDF document that can be selectively viewed or hidden by document authors or consumers. This capability is useful in items such as CAD drawings, layered artwork, maps, and multi-language documents"

## 1.3 PROBLEM DEFINITION

Our objective is to build an API that allows users to

- Create animations for algorithms already implemented by giving different input data.
- Implement her own algorithms and create animations for those algorithms.

## 1.4 ORGANIZATION OF THE REPORT

A brief overview of existing tools and description of few other technologies that can be used to create such an API are given in Chapter 2. Chapter 3 describes the proposed methodology and the architecture of the levels of the API. Chapter 4 explains the results achieved with relevant screenshots, and the impact of the tool on pedagogy. In Chapter 5, we list a few possible future enhancements.

# CHAPTER 2
# LITERATURE REVIEW

## 2.1 METAPOST

John Hobby's MetaPost is arguably the first graphics language to enable drawing quality diagrams in TEX. MetaPost is a batch-oriented graphics language based on Knuth's MetaFont, but with PostScript output and numerous features for integrating text and graphics. MetaPost is a powerful language for producing figures for documents to be printed on PostScript printers, either directly or embedded in TEX documents. MetaPost is able to integrate text and mathematics, marked up for use with TEX, within the graphics.

## 2.2 LUA

Lua is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting byte code for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Lua has a deserved reputation for performance. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. Substantial fractions of large applications have been written in Lua.

Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it.

## 2.3 ASYMPTOTE

Asymptote is a powerful vector graphics language that provides a natural coordinate- based framework for technical drawing. Labels and equations are typeset with LaTeX, for high-quality PostScript output. A major advantage of Asymptote over other graphics packages is that it is a full-blown programming language, as opposed to just a graphics program. The salient features of Asymptote are that it

- Provides a portable standard for typesetting mathematical figures, just as TeX/LaTeX has become the standard for typesetting equations;
- Generates high-quality PostScript, PDF, SVG, or 3D PRC vector graphics;
- Embeds 3D vector PRC graphics within PDF files
- Runs on all major platforms (UNIX, MacOS, Microsoft Windows);
- Typesets labels in LaTeX (for document consistency);
- Uses simplex method and deferred drawing to solve overall size constraint issues between fixed-sized objects (labels and arrowheads) and objects that should scale with figure size;
- Fully generalizes MetaPost path construction algorithms to three dimensions;
- Implements high-level graphics commands in the Asymptote language itself, allowing them to be easily tailored to specific applications.

## 2.4 PGF/TikZ

PGF/TikZ is a tandem of languages for producing vector graphics from a geometric algebraic description. PGF is a lower-level

language, while TikZ is a set of high-level macros that use PGF. The top-level PGF and TikZ commands are invoked as TEX macros, but in contrast with PSTricks, the PGF/TikZ graphics themselves are described in a language that resembles MetaPost. The interpreter for PGF and TikZ is written in TEX.

## 2.5 AUTOMATIC LAYOUT OF GRAPHS

In Drawing Graphs with MetaPost, John Hobby explains about a graph-drawing package that has been implemented as an extension to the MetaPost graphics language. MetaPost has a powerful macro facility for implementing statistical graphs. There are also some new language features that support the graph macros. Existing features for generating and manipulating pictures allow the user to do things that would be difficult to achieve in a stand-alone graph package.

Jannis Pohlmann proposed Configureable Graph Drawing Algorithms for the TikZ Graphics Description Language. It provides an introduction to the graph drawing features of TikZ, a versatile language for creating a wide range of vector graphics that includes a framework for automatic graph layout strategies written in the Lua programming language. Algorithms for two families of drawing methods — spring and spring-electrical layouts of general graphs as well as layered drawings of directed graphs were developed specifically for TikZ. They are presented along with a discussion of their underlying concepts, implementation details, and options to select, configure, and extend them. The generated spring and spring-electrical layouts feature a high degree of symmetry, evenly distributed nodes and almost uniform edge lengths. The techniques developed for layered drawings are successful in highlighting the flow of directed graphs and producing layouts with uniform edge lengths and few

crossings.

Konstantin Skodinis proposed construction of linear tree-layouts which are optimal with respect to vertex separation in linear time, in which he presents a linear time algorithm which, given a tree, computes a linear layout optimal with respect to vertex separation.

### 2.5.1 FORCE DIRECTED GRAPH DRAWING

Force-directed graph drawing algorithms are a class of algorithms for drawing graphs that are aesthetically pleasing. The objective is to position the nodes of a graph in two-dimensional or three-dimensional space so that all the edges are of more or less equal length and there are as few crossing edges as possible; this is achieved by assigning forces among the set of edges and the set of nodes, based on their relative positions, and then using these forces either to simulate the motion of the edges and nodes or to minimize their energy.

While graph drawing can be a difficult problem, force-directed algorithms, being a physical simulation, usually require no special knowledge about graph theory such as planarity.

## 2.6 T-ANIMATION MODULE

The t-animation package provides an interface to create portable, JavaScript-driven PDF animations from sets of vector graphics or rasterized image files or from inline vector graphics, such as LaTeX picture, PSTricks or PGF/TikZ generated pictures, or just from typeset text. It supports the usual PDF making workflow. The resulting PDF can be viewed in current Adobe Readers on all supported platforms. t-animation module is especially useful for animating a diagram as a sequence of frames. It provides forward, backward, pause and resume controls.

## 2.7 TYPESETTING LANGUAGES

### 2.7.1 TEX

TEX is a typesetting system and allows anybody to produce high-quality books using a reasonably minimal amount of effort, and to provide a system that would give exactly the same results on all computers. TEX is a popular means by which to typeset complex mathematical formulae; it has been noted as one of the most sophisticated digital typographical systems in the world. TEX is popular in academia, especially in mathematics, computer science, economics, engineering, physics, statistics, and quantitative psychology.

### 2.7.2 ConTEXt

ConTEXt is a general-purpose document processor. It is especially suited for structured documents, automated document production, very fine typography, and multi-lingual typesetting. It is based in part on the TEX typesetting system, and uses a document markup language for manuscript preparation. The typographical and automated capabilities of ConTEXt are extensive, including interfaces for handling micro typography, multiple footnotes and footnote classes, and manipulating OpenType fonts and features. Moreover, it offers extensive support for colors, backgrounds, hyperlinks, presentations, figure-text integration, and conditional compilation. It gives the user extensive control over formatting while making it easy to create new layouts and styles without learning the low-level TEX macro language.

As its native drawing engine, ConTEXt integrates a superset of MetaPost called MetaFun which allows the users to use the drawing abilities of MetaPost for page backgrounds and ornaments. Metafun can also be used with stand-alone MetaPost. ConTEXt also supports the use of other external

drawing engines, like PGF/TikZ and PSTricks.

It also allows the user to use different TEX engines like pdf TEX , Xe TEX and Lua TEX without changing the user interface.

### 2.7.3 LuaLATEX

LuaLATEX is a TEX -based computer typesetting system which started as a version of pdf TEX with a Lua scripting engine embedded. After some experiments it was adopted by the pdf TEX team as a successor to pdf TEX (itself an extension of eTEX , which generates PDFs). Later in the project some functionality of Aleph was included (esp. multi-directional typesetting). The main objective of the project is to provide a version of TEX where all internals are accessible from Lua. In the process of opening up TEX much of the internal code is rewritten. Instead of hard coding new features in TEX itself, users (or macro package writers) can write their own extensions. Lua LATEX offers native support for OpenType fonts.

# CHAPTER 3

# DESIGN AND IMPLEMENTATION

## 3.1 FEATURES

Our proposed system is an interface (set of API's) for laying out diagrams for data structure such as arrays, lists, trees and graphs, and for animating the algorithms that operate on them such as insertion of nodes in a binary search tree. The API can be used by a ConTEXt user to produce PDF documents with animated diagrams. The animation is a sequence of frames where each frame contains a data structure diagram generated by the layout module. Frames could be created in such a fashion that they show each step of the algorithm. In a frame, the node(s) or element(s) that are currently being processed or modified could be highlighted with a color chosen by the user. Once the animation is created and embedded into a PDF document, it can be played, paused, or moved around in any fashion.

In the animation, breakpoints (the most important frames) can be set and the animation can be made to automatically pause at breakpoints. It might be intuitive to animate only through the breakpoints while teaching certain algorithms. The tool allows the user to set breakpoints while designing the animation; while viewing the document, the viewer can turn on or off the use of breakpoints by pressing a toggle button in the animated diagram. If the use of breakpoint are on, the animation will pause at these points; otherwise, the animation will proceed as though there are no breakpoints.

In addition to displaying the diagrams, the pseudocode for the corresponding diagram, with the currently executing line highlighted, is also displayed above, thus making it easy for the viewer to understand the operation of algorithm.

## 3.2 IMPLEMENTATION FRAMEWORK

The overall architecture of the system is shown in Figure 1.

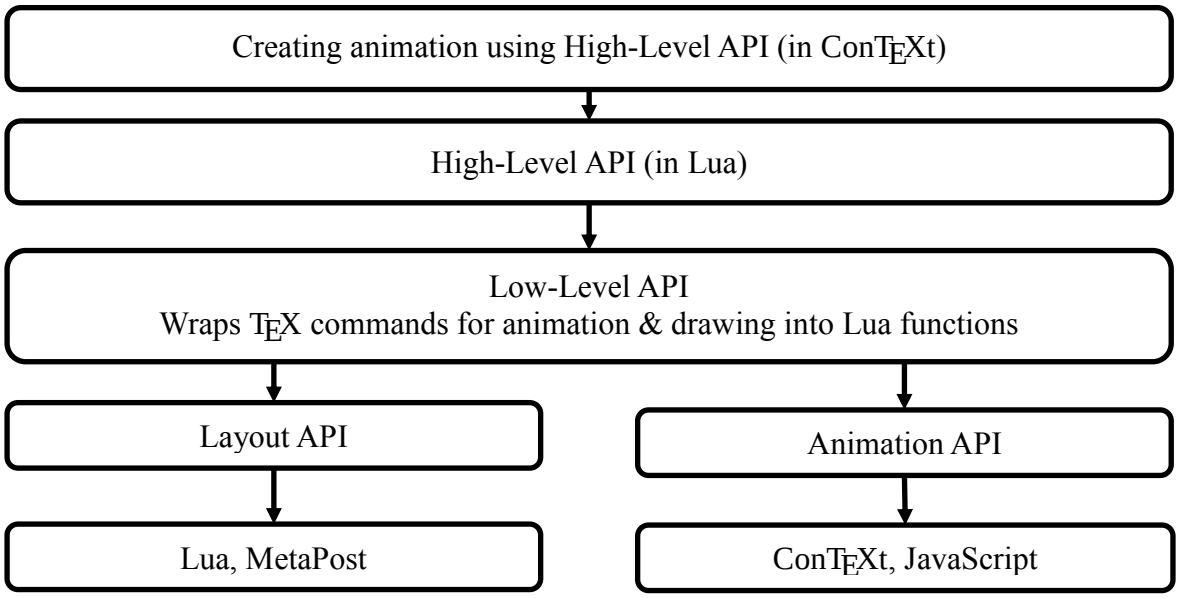


**Figure. 1: Layers in the Interface**

The interface provides a comprehensive multi-level API (low-level and high-level) that would enable the users to generate an entire algorithm animation and embed it to a PDF document in a single function call. The API's are written in Lua. The low-level API acts as an interface between the high-level API and the underlying drawing and animation subsystems. We use MetaPost for drawing and a modified version of ConTEXt's third party t-animation module for animation. The major work involves creating and standardizing the low-level API.

To use these levels of API, we envisage two classes of users as indicated in Figure 2, namely expert (pro) and end users. End users do simple things such as using the algorithms that are created by the expert users using our interface and it must be easy for them to use the interface. They use the high-level API to create animations. Expert users can do complex things such as creating new animations for algorithms and it must be possible for them to do.

They use the low-level API to create new animation algorithms and pack similar algorithms into a high-level API.

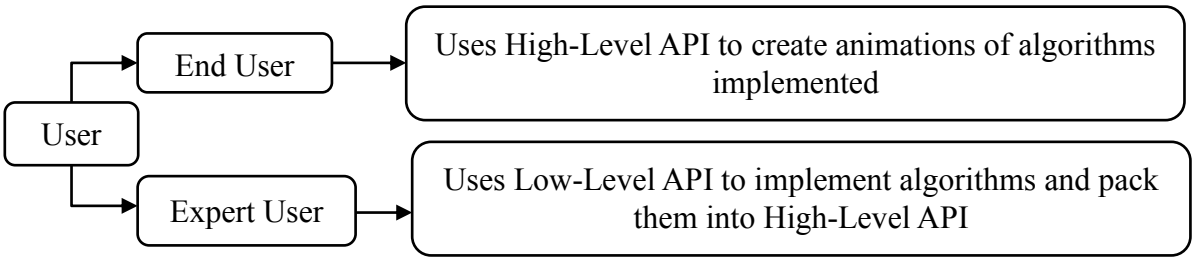


**Figure. 2: User Classification**

### 3.2.1 LOW-LEVEL API

To handle the two important tasks of layout and animation, the system has two separate, flexible API's.

- Layout API – This provides a set of functions to draw variety of data structure diagrams involving arrays, lists, trees and graphs.
- Animation API – This has the functions necessary for animating the diagrams generated with the layout API.

#### 3.2.1.1 LAYOUT API

MetaPost is the underlying language that is responsible for drawing the diagrams. It provides ways to draw elementary shapes like circles, rectangles, ovals, lines and text. Colors can be given as well. To use MetaPost to create data structure diagrams, we first have to combine certain MetaPost commands and wrap them in Lua functions that can be used directly in algorithm animations. For example, to draw trees, in the co-ordinate space, the nodes are constructed in such a way that, once the X-Y co-ordinates and label are passed to a Lua function, it will draw a circle with the label at the center of it.

```
function drawNode(x,y,num,nodeColor,labelColor,radius,textSize)
       context(
       "drawfill("..(x-
       radius)..","..y..")..("..x..","..(y+radius)..")..("..
       (x+radius)..","..y..")..          ("..x          ..","..(y-
       radius) ..")..cycle withcolor "..nodeColor..";"
       )
       context("defaultscale := "..textSize..";")
       context("label(\""..num.."\",    ("..x    ..","..y    .."))
       withcolor "..labelColor..";")
end
```

In this way, all required functions are constructed before proceeding to write the layout algorithms for data structures. This forms the basic drawing API. Using this basic API, the algorithms are written.

## ARRAYS

Drawing arrays is straightforward. Each element has to be juxtaposed to the next element to form the structure of an array. Elements are basically boxes with text inside them. Each box is associated with a name and they can be accessed using that name. Before any element is drawn, it has to be specified that each element's north-east point and south-east point must meet the next element's north-west and south-west point respectively.

```
b1.ne = b2.nw
b1.se = b2.sw
```

These equations will make the boxes cling to each other thus forming an array structure. To get a proper array representation, the array indices have to be shown. The indices are displayed on top of each element. This is done by creating a label and making the south point of the label meet the north point of the corresponding box.

```
l1.s = b1.n
```

This gives us a proper array representation but this might not be enough to make viewers understand the algorithm. To show the proceedings of an algorithm in a better way, we have added an extra element called a *marker*.

Markers are vertical lines with a text at the bottom that are designed essentially to indicate the array value that is currently stored in a variable. In Figure 3, the value of 23 is stored in the `Min` variable and the index of the array element is 6.

Background color and label color for each box can be specified.

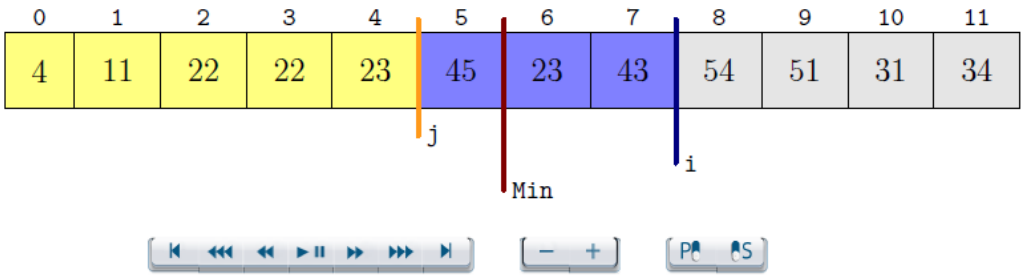


**Figure. 3: Arrays**

## LISTS

Lists consist of nodes that are linked with pointers (an arrow head). Nodes of the linked list can be represented in two ways. One way is to represent them as circles with a label inside them and a pointer to the next node. The other way is to represent them with two joined rectangles in each node with first rectangle having the label and next rectangle having a pointer to the next node. The representation with rectangles is useful when the viewer is new to the concept of lists. The box-and-arrow representation is more intuitive for a beginner and can be easily related to the actual mechanism of lists. Once the viewer is familiar with linked lists, the circular representation could be used --- it is more abstract; it occupies less space than the box-and-arrow representation and thus allows more nodes to be shown clearly.

Base API functions for both cases have to be developed and have to be called depending on which representation is required for that particular algorithm.

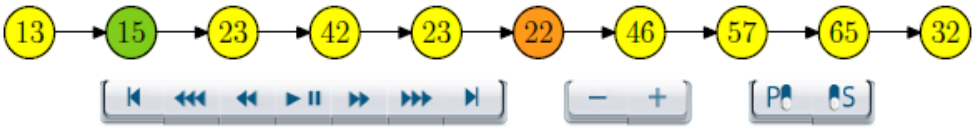


**Figure. 4: Lists – Abstract Representation**

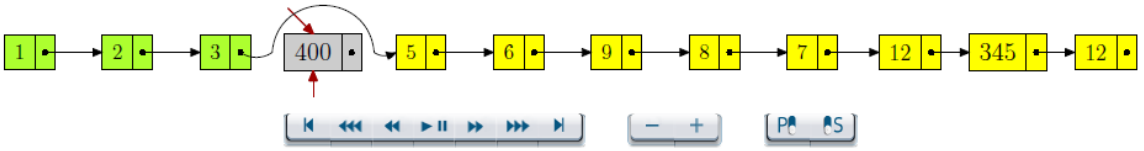


**Figure. 5: Lists – Box-and-arrow Representation**

## TREES

Trees can range from conventional binary trees to more complex trees like threaded binary trees. We have implemented the layout for simple trees. The basic step for drawing binary trees is that, with respect to a parent node, the two child nodes must be spaced out equally in positive and negative X axis and they must be placed below the parent node.

The spacing of the nodes is the major concern in drawing trees. As the depth increases, the nodes must be spaced out accordingly. Not only must the spacing out be adjusted to accommodate the tree inside the page but also the scaling factor must be changed. Scaling issue is addressed in the subsequent section where the layout of graphs is discussed.

The algorithm generates a tree given a set of nodes with the links to their child nodes.

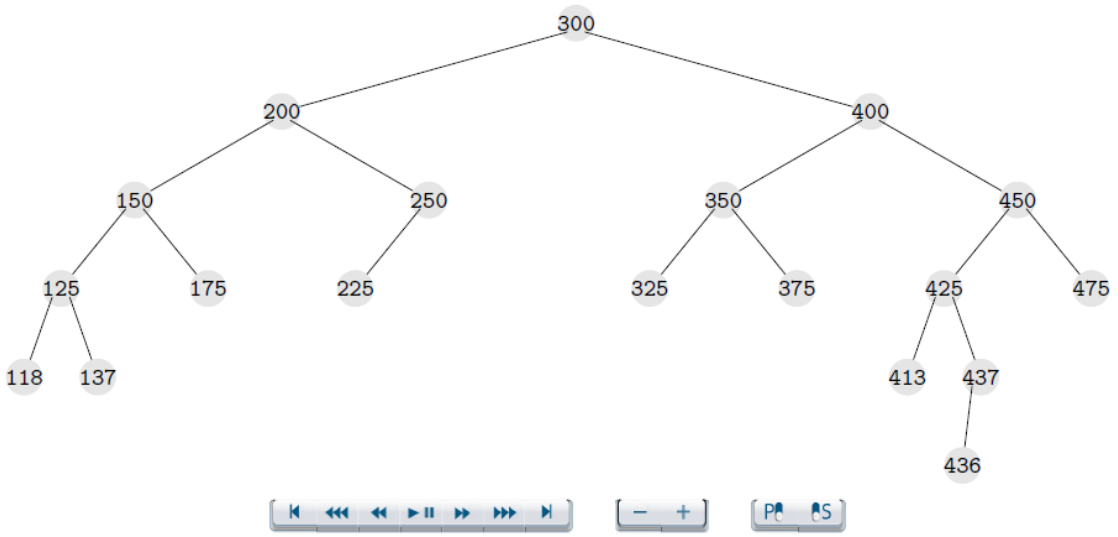


**Figure. 6: Trees**

## GRAPHS

Graphs are frequently drawn as node-link diagrams in which the nodes are represented as disks or boxes and the edges are represented as line-segments. Graph-Drawing is in itself a separate branch of research. We use one of the most famous graph-drawing methods to draw the graph, the *Force Directed Graph Drawing* Algorithm. This assigns forces among the set of edges and the set of nodes of a graph drawing. Typically, spring-like attractive forces based on Hooke's law are used to attract pairs of endpoints of the graph's edges towards each other, while simultaneously repulsive forces like those of electrically charged particles based on Coulomb's law are used to separate all pairs of nodes. In equilibrium states for this system of forces, the edges tend to have uniform length (because of the spring forces), and nodes that are not connected by an edge tend to be drawn further apart (because of the electrical repulsion). Edge attraction and vertex repulsion forces may be defined using functions that are not based on the physical behavior of springs and particles; for instance, some force-directed systems use springs whose attractive force is logarithmic rather than linear.

Using this algorithm, we were able to generate aesthetically pleasing

graphs with less crossings and almost equal edge length. The algorithm supports layout of both directed and undirected graphs. Having laid out the graph, the next issue is labeling the edges. The labels should be placed in accordance with the slope of the edges. The algorithm takes adjacency matrix as input to generate the graphs.

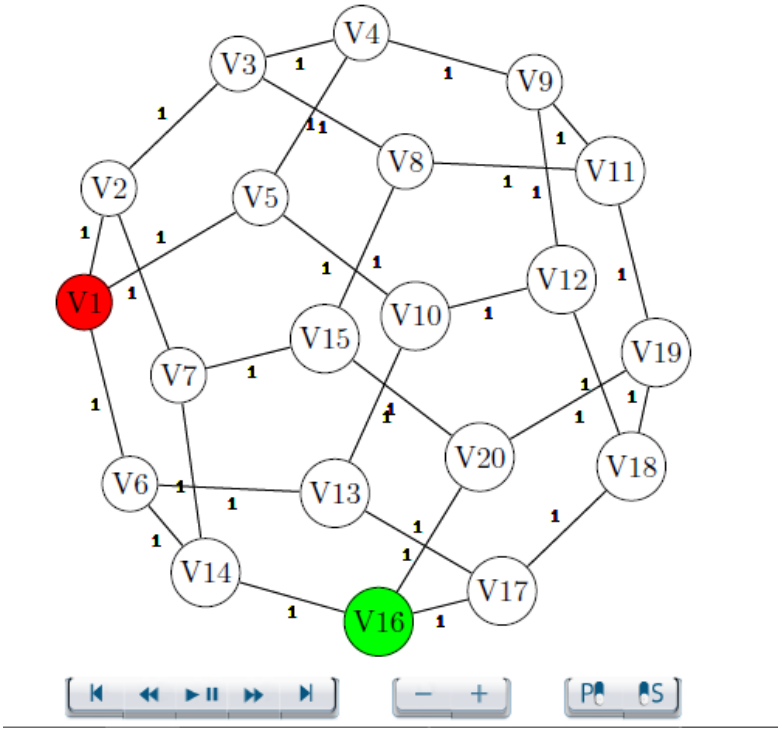


**Figure. 7: Graphs**

## SCALING

One common problem faced in laying out of any of these data structures is that the diagrams have to be accommodated within the PDF document regardless of the number of nodes in the diagram. For instance, in Arrays, the diagram will go out of the boundary when there are more than 50 elements and the readability of the labels will also decrease at that stage. The same applies for other data structures. There is one straightforward way of approaching this problem. The scaling factor must be reduced proportional to the number of nodes.

$$Scaling\ factor\ \alpha\ \frac{1}{No.of\ Nodes}$$

But this is not the most elegant solution. More nodes mean more number of frames. For an algorithm like selection sort, with 80 elements, the normal animation would take more than 10 minutes for completion which is meaningless. In those cases, it would be more intuitive to automatically increase the frame rate of the animation and with 80 elements no label would be visible. To overcome this issue, we draw large dots when the number of elements exceed a threshold value. The threshold depends on the data structure. From the tests we have done, we could determine the threshold for each of the data structures and it is given in table.

| Data structure | Threshold (in number of nodes) | |
|---|---|---|
| Arrays | 80 | |
| List | Abstract | Box-and-arrow |
| | 80 | 60 |
| Trees | 6 (Depth of tree) | |
| Graph | 50 | |

**TABLE 1: Threshold for scaling down**

### 3.2.1.2 ANIMATION API

The API is structured as described in the below diagram.

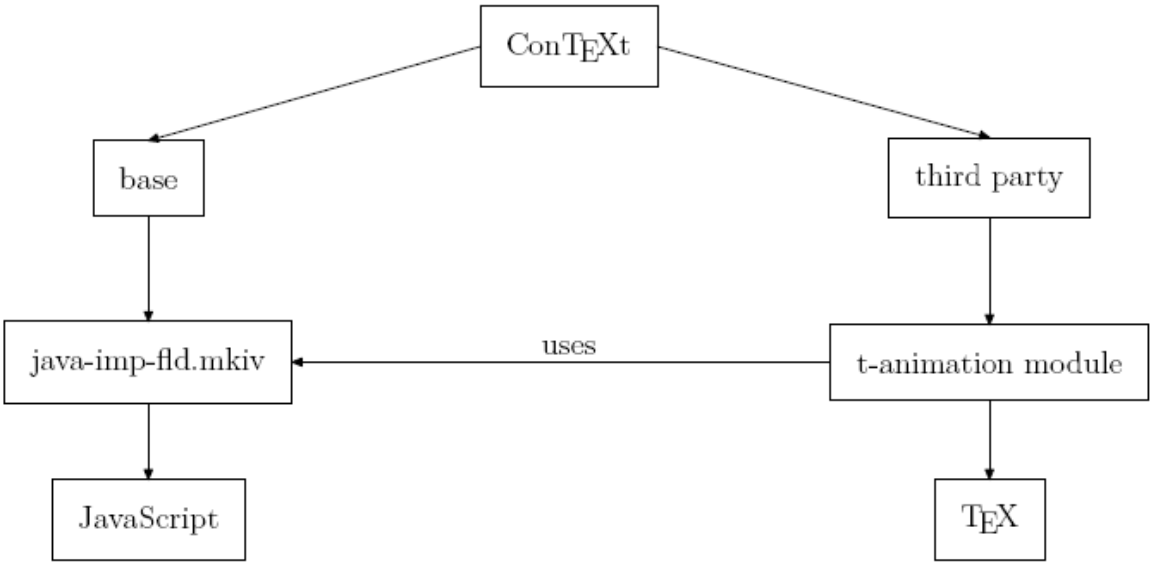


**Figure. 8: Architecture of t-animation module**

Animations are created by having frames in the same location and toggling the visibility of frames in a particular order to get the effect of an animation. In ConTEXt, this is done using the fieldstack mechanism which uses JavaScript to traverse the frames in order. This JavaScript file (java-imp-fld.mkiv) is part of the core ConTEXt distribution. This is crude and using this for actual animation could be a tedious task. The third-party open-source *t-animation* module provides an easy to use interface for this fieldstack mechanism to create animations. This module is written in ConTEXt and it can be considered as a wrapper to the JavaScript functions which provides an interface that allows users to create animations.

It has basic functions like play, pause, go to previous and next frames. These are accessible through buttons on the PDF document. But we need a more robust animation module with more useful and flexible features.

Frame is the most basic element of an animation. When more properties are added to a frame, more features could be added to the animation module. We have added a priority level for each frame ranging from 0 to 2.

- Level 0 – *Sequential Frames***,** the less important frames.
- Level 1 – *Normal Frames*
- Level 2 – *Breakpoints***,** the most important frames.

Once these are set, this information can be used to provide more useful options. When the animation explains a recursive algorithm, the most important steps are in the outer-loop and so, they could be the breakpoints. All the frames generated in the inner-loop are less important and thus could be the sequential frames. While explaining recursive algorithms, at certain times, it would be convenient to pause the animation once the breakpoints are reached and at certain times, skipping the sequential frames would prove useful. We have added both these features. The animation can be made to **automatically pause at the breakpoints** and it can be made to **skip sequential frames** or even both at the same time. These options are off by default and can be enabled or disabled by a button click in the animation. When the presenter manually chooses to move around the animation, in addition to the usual 'go to next frame' and 'go to previous frame' buttons, we also have the 'go to next breakpoint' and 'go to previous breakpoint' buttons thus making the most use of the priority levels.

*Frame rate* is the frequency at which animation displays unique consecutive images called frames. Larger the frame rate, greater is the speed of the animation.

$$frame\ rate\ \boldsymbol{\alpha}\ speed\ of\ animation$$

We provide a function that can be used to set the default frame rate of an algorithm animation. The frame rate of an animation can also be increased or decreased when the animation is being played by using the buttons in the animation.

Since our API is in Lua, the features that we have added to t-animation module must be made accessible through Lua functions. We have a Lua wrapper over the t-animation functions which exposes all possible t-animation capabilities as Lua functions which can be used during algorithm creation.

**setupAnimation(frameRate)** – The animation can be setup with the required frame rate.

**startAnimation(menu, repeat)** – This will start the animation with the options specified. Menu can be made visible or invisible. The animation can be made to run on loop or be made to stop after one run.

**startFrame(importanceLevel)** – A frame can be started and the frame can be made as a breakpoint or a sequential frame by passing integer arguments to the corresponding parameters.

### 3.2.2 HIGH-LEVEL API

The high-level API has a set of functions in Lua that animate the algorithms on predefined data structures. This level of API is open to additions from the Pro-Users. The Pro-Users can implement their own algorithms and create animations for the same by using the low-level API. We provide along with our package a few examples of algorithm animation creation functions for each data structure to make the users understand how to use our API efficiently.

Creating an animation is as simple as adding a few lines to the original algorithm.

**Steps to create an algorithm animation**

1. Write the algorithm in lua
2. Add the startAnimation() call with all required options to the first line of algorithm
3. Add endAnimation() call before the function ends

4. Whenever a new frame has to be created with a new diagram, start a frame using the startFrame() function.
5. Set options for the diagram such as colors and markers, and call the lower-level API's draw function to draw the diagram.
6. Close the frame with a call to stopFrame() function.

**Outline of an animated algorithm**

```
function newAlgorithm()
     startAnimation(options)
     …
     for(…)
          …
          startFrame()
          beginPicture()
          …
          drawArray(options)/drawTree(options)/
          drawGraph(options)/drawList(options)
          …
          endPicture()
          stopFrame()
          …
     end
     …
     stopAnimation()
end
```

# CHAPTER 4

# EVALUATION AND RESULTS

## 4.1 RESULTS ACHIEVED

We have created a standard and efficient API for the layout and animation of data structure diagrams like arrays, lists, trees and graphs. Sophisticated diagrams can also be drawn with ease.

### 4.1.1 BASICS OF REPRESENTING DATA STRUCTURES

Individual data structure keys need to be represented either inside a box (in case of arrays, for example) or inside a circle (in case of trees). For this purpose we use the 'boxes.mp' MetaPost library. The library on its own does not allow much customization. Hence we created a modified version of the library to allow users to customize their graphic representations.

### 4.1.2 CREATING A SERIES OF BOXES

In situations like representing arrays, a series of boxes need to be created. Each box must have a unique name. Once the boxes are created, the alignment of boxes with respect to each other must also be mentioned.

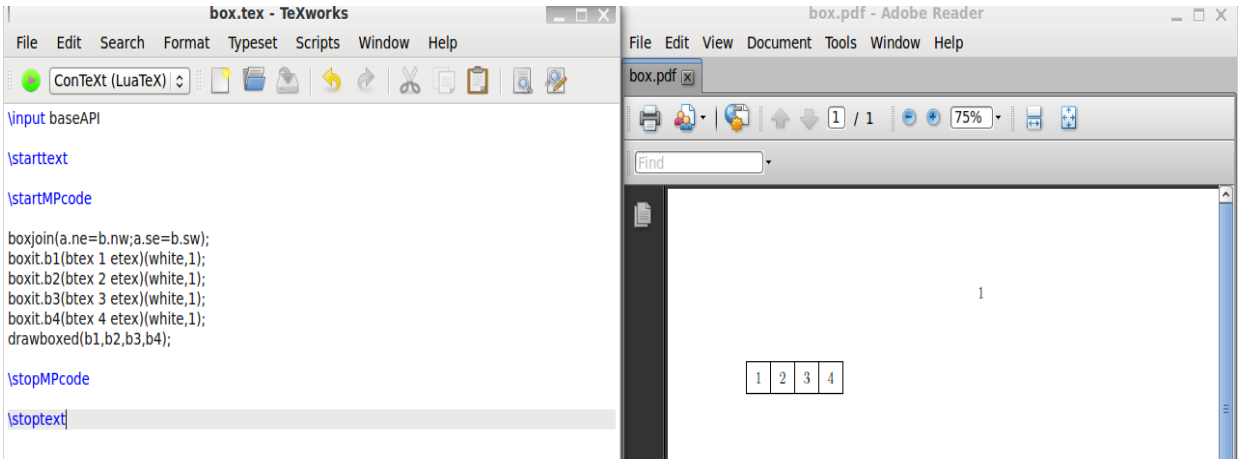


**Figure. 9: Creating an array with boxes**

### 4.1.3 CREATING CIRCLES

Circles are used in representation of lists, trees and graphs. For drawing a circle, the center point needs to be given in (x, y) co-ordinate pairs.

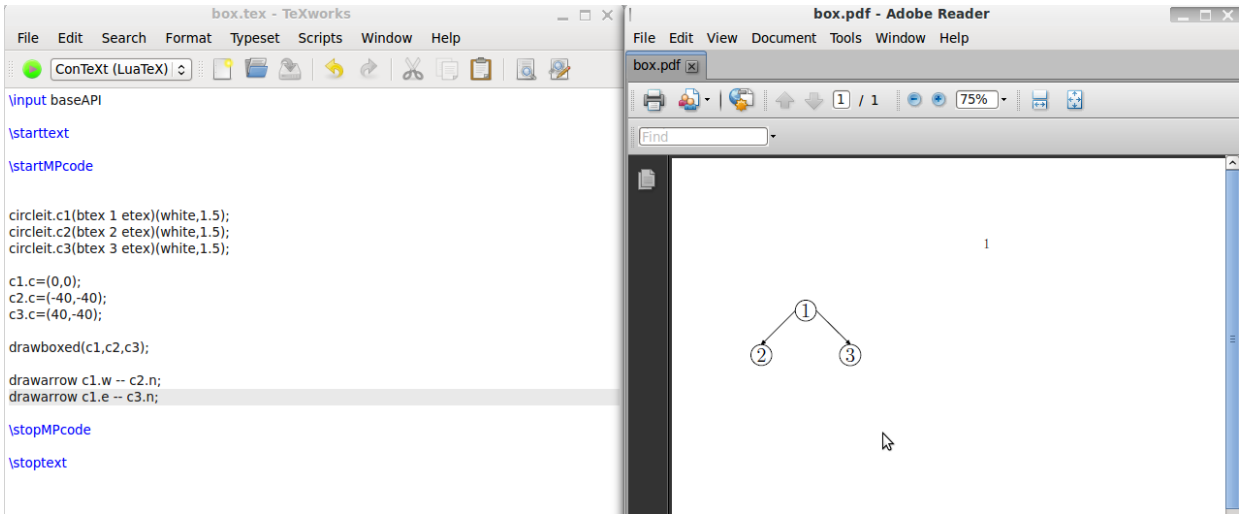


**Figure. 10: Creating tree with circles**

### 4.1.4 CREATING FRAMES

Animations are a series of frames displayed in a sequence. When creating an algorithm animation, the diagrams that are drawn between *\frame{* and *}* are drawn in one frame. So, whenever a frame is needed, start a frame using *\frame{*, draw necessary diagrams and close it with a *}*.

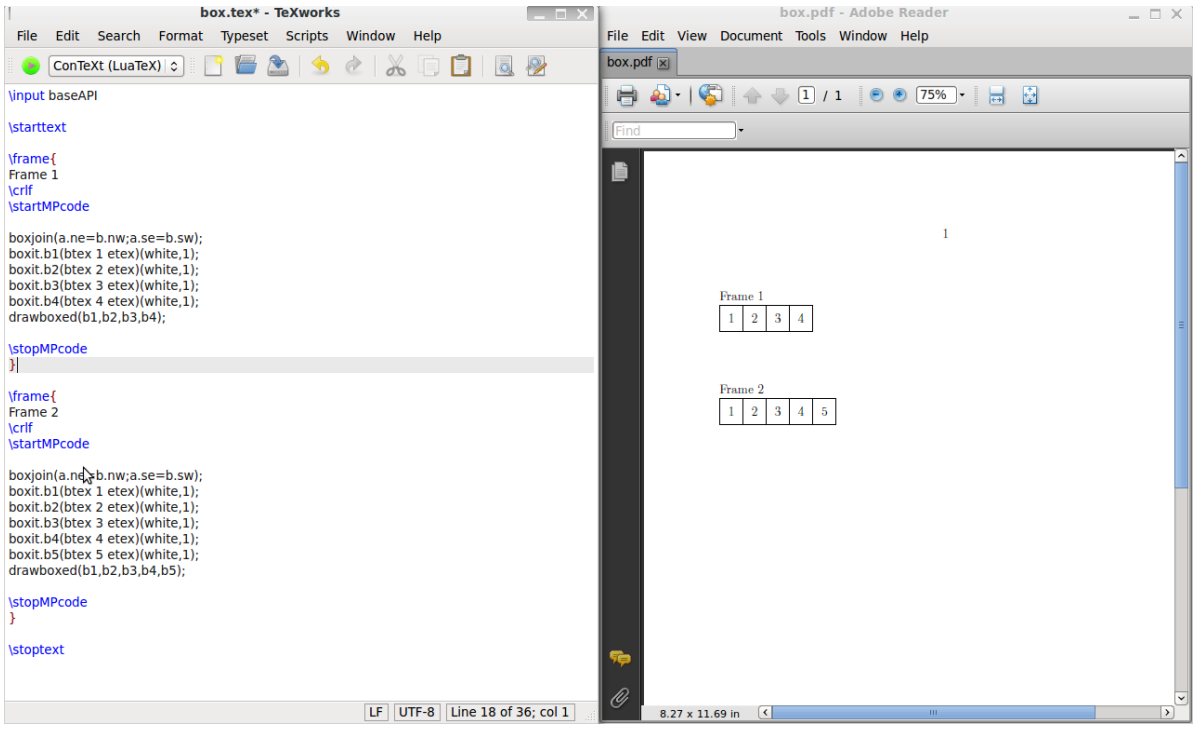


**Figure. 11: Frame creation**

### 4.1.5 GENERATING THE ANIMATION

Once the frames are created, the animation is generated automatically, with all the necessary controls. Customizations are also available in the animations like altering frame rate, skipping frames, etc. Controls for animation are available in the animation.

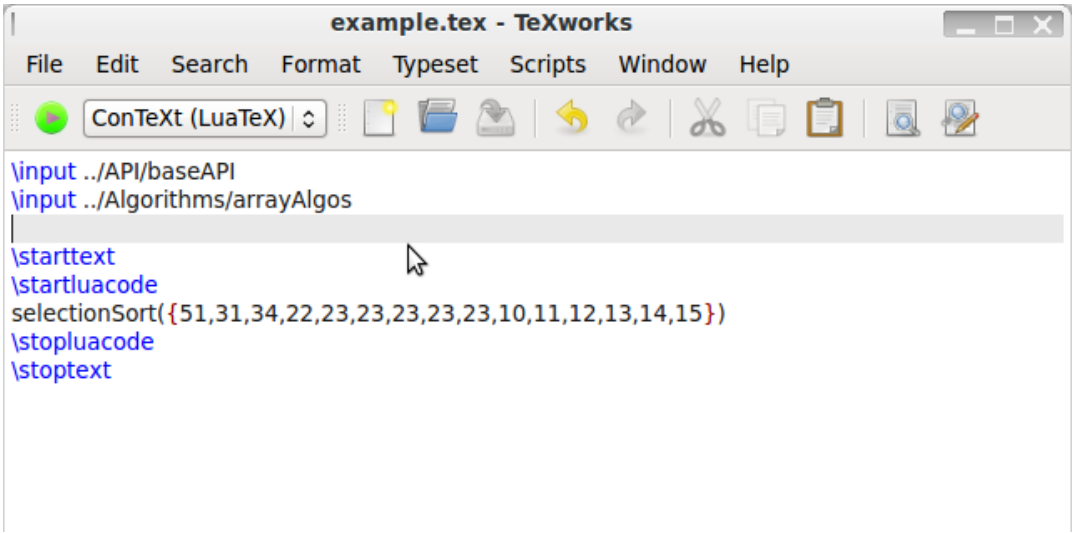


**Figure. 12: Function call**

```
for j=1,n,1 do
    iMin = j
    for i = j+1,n,1 do
        if a[i] < a[iMin] then
            iMin = i
        end
    end
    swap(j,iMin)
end
```

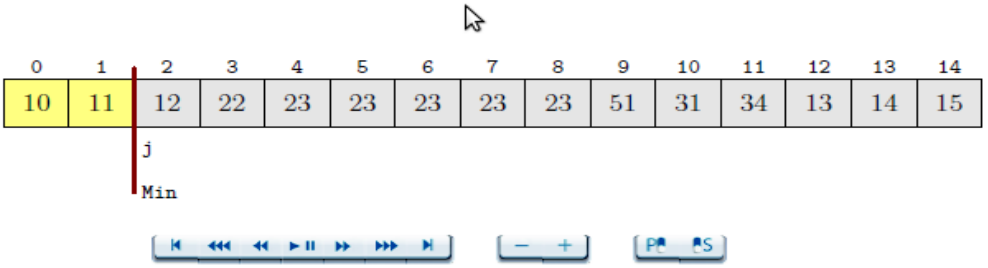


**Figure. 13: Animation frame 1**

```
for j=1,n,1 do
    iMin = j
    for i = j+1,n,1 do
        if a[i] < a[iMin] then
            iMin = i
        end
    end
    swap(j,iMin)
end
```

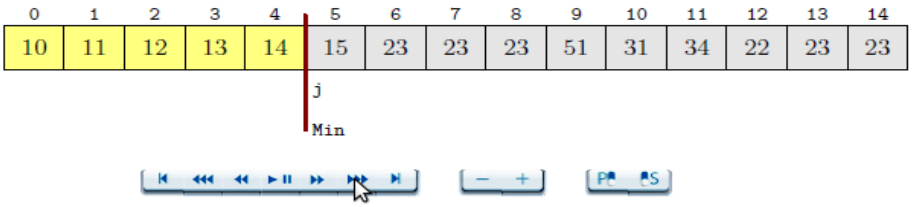


**Figure. 14: Animation frame 2**

### 4.1.6 WRITING NEW ALGORITHMS

The user may be required to write algorithms on his own. This is absolutely possible with our API. The user needs to include the necessary API files in his working file and call the necessary functions to draw diagrams wherever she wishes as shown in the sample below

```
function bstCreate(a, frameRate)
    bstCreate1(a)
    elements = {}
    root = nil
    setupAnimation(100)
    startAnimation("yes","no")
    loadTreeDefaults()
    fixTreeHeight(4)
    for i=1,#a do
        elements[i]={}
        elements[i].key=a[i]
        elements[i].left={}
        elements[i].right={}
        elements[i].level=0
        elements[i].parent=0
        if i~=1 then
            local depth = maxTreeHeight(root)
            startFrame()
            beginPicture()
            drawTreeWithArray(root, depth, "1")
            endPicture()
            stopFrame()
        end
        insert(elements[i])
        if i==1 then
            local depth = maxTreeHeight(root)
            startFrame()
            beginPicture()
            drawTreeWithArray(root, depth, "1")
            endPicture()
            stopFrame()
        end
    end
    local depth = maxTreeHeight(root)
    startFrame()
    beginPicture()
    showTreePseudoCode(1,5)
    drawTreeWithArray(root, depth, "1")
    endPicture()
```

```
    stopFrame()
    stopAnimation()
end
```

## 4.2 IMPACT ON PEDAGOGY

We showed the animations of data structures created using the API to a group of Computer Science students for their feedback and many expressed that it made it easy to understand the data structures better. Also the pseudo code being displayed alongside the animation drew a lot of attention. Our API can be used to create animations where TEX is used to generate presentations for lectures. The PDF documents can be viewed using Adobe reader, a freeware. However, evaluating the use and impact of the tool more quantitatively in a teaching-learning environment still remains to be carried out.

# CHAPTER 5

# CONCLUSION AND FUTURE DIRECTION

## 5.1 CONCLUSION

An API has been created using Lua and MetaPost. It provides a quick and clean way of creating animations of commonly used data structures. Many commonly used algorithms are bundled with the API for which the user can create animations with a single line of code. It also allows the users to write their own algorithms and create animations for those algorithms with minimum knowledge of Lua. The tool uses open standards and is built using freely available technologies. The PDF documents can be viewed on any system with adobe reader version 6.0 or above. The paradigm and the tool have been experimented in small scale with fair feedback from trials, and have proved the promise it holds for laying out the data structures effectively and creating the animations.

## 5.2 FUTURE ENHANCEMENT

Although the API in its current form is a useful tool, there is scope for a few more features that are outlined below, to make it a complete and powerful tool:

- A general improvement would be to make the layout algorithm more robust and support many other data structures.
- The layout for lists supports only linear linked lists now. This could be improved to support circular lists.
- Tree layout algorithm supports only up to two children for a parent. So, algorithms on trees like B-Trees cannot be implemented with the current API. So, the algorithm can be extended to support multiple children for a parent.
- We have added very useful features to the t-animation module. With the

features that we have added, our module could be extended to a full-fledged presentation library.

## APPENDIX 1

## MODIFICATION OF BOXES.MP

Boxes.mp is a MetaPost library for drawing boxes, circles and it makes it easy to place the boxes properly and connect them whenever needed. The library is written in MetaPost. To use this in our API, we needed it to be more feature-rich. We needed options to set color for the boxes like draw color, fill color, label color and for using it in arrays, we needed to draw markers to indicate the proceedings of the algorithm. To fulfill our needs, we have modified the boxes.mp library and used it in our API creation.

```
% Macros for boxes
def drawboxed(text t) =          % Draw each box
  fixsize(t); fixpos(t);
  forsuffixes s=t: drawfill bpath.s withcolor s.col;draw
pic_mac_.s; draw bpath.s; endfor
enddef;

def fillboxed(text t) =          % Draw each box
  fixsize(t); fixpos(t);
  forsuffixes s=t: drawfill bpath.s withcolor s.col;draw
left_pic_mac_.s; endfor
enddef;

def drawMarker(text t)(expr indH,indVal,indColor) =
  forsuffixes s=t: draw s.nw+(0,5) -- s.sw-(0,indH) withpen
pencircle scaled 1.5pt withcolor indColor;label.rt(indVal,s.sw-
(0,indH)); endfor
enddef;

def drawunboxed(text t) =        % Draw contents of each box
  fixsize(t); fixpos(t);
  forsuffixes s=t: draw pic_mac_.s ; endfor
enddef;

def drawleftunboxed(text t) =        % Draw contents of each box
  fixsize(t); fixpos(t);
  forsuffixes s=t: draw left_pic_mac_.s ; endfor
enddef;



vardef boxit@#(text tt)(expr cellColor,scaleVal) =
```

```
  beginbox_("boxpath_","sizebox_",scaleVal,@#,tt);
  generic_declare(pair) _n.sw, _n.s, _n.se, _n.e, _n.ne, _n.n,
_n.nw, _n.w;
    generic_declare(color) _n.col;
  @#col := cellColor;
  0 = xpart (@#nw-@#sw) = ypart(@#se-@#sw);
  0 = xpart(@#ne-@#se) = ypart(@#ne-@#nw);
  @#w = .5[@#nw,@#sw];
  @#s = .5[@#sw,@#se];
  @#e = .5[@#ne,@#se];
  @#n = .5[@#ne,@#nw];
  @#ne-@#c = @#c-@#sw = (@#dx,@#dy) + .5*(urcorner pic_@# -
llcorner pic_@#);
  endbox_(clearb_,@#);
enddef;

vardef circleit@#(text tt)(expr cellColor,scaleVal) =
  beginbox_("thecirc_","sizecirc_",scaleVal,@#,tt);
  generic_declare(pair) _n.n, _n.s, _n.e, _n.w,_n.ne, _n.sw,
_n.se, _n.nw;
  generic_declare(color) _n.col;
  @#col := cellColor;
  @#e-@#c = @#c-@#w = (@#dx,0) + .5*(lrcorner pic_@#-llcorner
pic_@#);
  @#n-@#c = @#c-@#s = (0,@#dy) + .5*(ulcorner pic_@#-llcorner
pic_@#);
  endbox_(clearc_,@#);
enddef;
```

## APPENDIX 2

## FORCE DIRECED GRAPH DRAWING ALGORITHM IN LUA

Force-directed graph drawing algorithms assign forces among the set of edges and the set of nodes of a graph drawing. Typically, spring-like attractive forces based on Hooke's law are used to attract pairs of endpoints of the graph's edges towards each other, while simultaneously repulsive forces like those of electrically charged particles based on Coulomb's law are used to separate all pairs of nodes. In equilibrium states for this system of forces, the edges tend to have uniform length (because of the spring forces), and nodes that are not connected by an edge tend to be drawn further apart because of the electrical repulsion. Edge attraction and vertex repulsion forces may be defined using functions that are not based on the physical behavior of springs and particles; for instance, some force-directed systems use springs whose attractive force is logarithmic rather than linear.

```
function ForceDirectedGraph(args,graph)
      -- local nodes = nodes or self.nodes
      local function applyForce(node1,force)
            node1.acceleration[1]=node1.acceleration[1]+force[1]
            node1.acceleration[2]=node1.acceleration[2]+force[2]
      end
      local function Coulomb_repulsion()
            for _, node1 in ipairs(graph.nodes) do
                  --iterate through every other node and sum forces
                  for _, node2 in ipairs(graph.nodes) do
                        if(node1.name ~= node2.name) then
                              local lenVec = {}
                              lenVec[1]=node1.vertex[1]-node2.vertex[1]
                              lenVec[2]=node1.vertex[2]-node2.vertex[2]
                              local len = length(lenVec)+0.1
                              local normVec = {}
                              normVec[1]=lenVec[1] * (1/len)
                              normVec[2]=lenVec[2] * (1/len)
                              local force = {}
      force[1]=(normVec[1]*args.repulsion)/(math.pow(len,2)*0.5)
      force[2]=(normVec[2]*args.repulsion)/(math.pow(len,2)*0.5)
                              applyForce(node1,force)
      force[1]=(normVec[1]*args.repulsion)/(math.pow(len,2)*-0.5)
```

```
      force[2]=(normVec[2]*args.repulsion)/(math.pow(len,2)*-0.5)
                              applyForce(node2,force)
                        end
                  end
            end
      end
      local function Hooks_attraction(k)
            k = k or 1
            for _, eachEdge in ipairs(graph.edges) do
                  local node1=Graph.getNode(eachEdge:getFromNodeName())
                  local node2=Graph.getNode(eachEdge:getToNodeName())
                  local vec1=node1.vertex
                  local vec2=node2.vertex
                  local lenVec = {}
                  lenVec[1]=vec2[1]-vec1[1]
                  lenVec[2]=vec2[2]-vec1[2]
                  local len = length(lenVec)
                  local displacement = eachEdge.length - len
                  local normVec = {}
                  normVec[1]=lenVec[1] * (1/len)
                  normVec[2]=lenVec[2] * (1/len)
                  local force = {}
                  force[1]=normVec[1]*k*displacement*-0.5
                  force[2]=normVec[2]*k*displacement*-0.5
                  applyForce(node1,force)
                  force[1]=normVec[1]*k*displacement*0.5
                  force[2]=normVec[2]*k*displacement*0.5
                  applyForce(node2,force)
            end
      end
      local function Attract_to_centre()
            for _, eachNode in ipairs(graph.nodes) do
                  local force = {}
                  force[1]=(eachNode.vertex[1]*-
1.0)*(args.repulsion/50.0)
                  force[2]=(eachNode.vertex[2]*-
1.0)*(args.repulsion/50.0)
                  applyForce(eachNode,force)
            end
      end
      local function Update_velocity(timestep)
            for _, eachNode in ipairs(graph.nodes) do
      eachNode.velocity[1]=(eachNode.velocity[1]+(eachNode.acceleratio
n[1]*timestep))*args.damping
      eachNode.velocity[2]=(eachNode.velocity[2]+(eachNode.acceleratio
n[2]*timestep))*args.damping
                  eachNode.acceleration = {0,0}
            end
      end
```

```
      local function Update_position(timestep)
            for _, eachNode in ipairs(graph.nodes) do
      eachNode.vertex[1]=eachNode.vertex[1]+(eachNode.velocity[1]*time
step)
      eachNode.vertex[2]=eachNode.vertex[2]+(eachNode.velocity[2]*time
step)
            end
      end
      local function Total_energy()
            local energy = 0.0
            for _, eachNode in ipairs(graph.nodes) do
                  local speed=length(eachNode.velocity)
                  energy = energy+(0.5*speed*speed)
            end
            return energy
      end
      --setting up local vars to be used later
      args.repulsion = args.repulsion or 400.0
      args.desiredEdgeLength = args.desiredEdgeLength or 1
      args.stiffness = args.stiffness or 400.0
      args.coulomb = args.coulomb or true
      args.hooks = args.hooks or true
      args.small_num = args.small_num or 1
      args.damping = args.damping or .50
      local total_kinetic_energy = 0
      local timestep = 0.03
      repeat
            Coulomb_repulsion()
            Hooks_attraction(args.stiffness)
            Attract_to_centre()
            Update_velocity(timestep)
            Update_position(timestep)
      until Total_energy() < 0.01
      return true
end
```

## REFERENCES


[1] Personal correspondence with Wolfgang Schuster, the author of t-animation module of ConTEXt.
E-mail: wolfgang.schuster@gmail.com

[2] Andy Hammerlindl, John Bowman, and Tom Prince. Asymptote: The Vector Graphics Language, Natural Sciences and Engineering Research Council of Canada, the Pacific Institute for Mathematical Sciences, and the University of Alberta Faculty of Science.

[3] Jannis Pohlmann. Configurable Graph Drawing Algorithms for the TikZ Graphics Description Language. Institute for Theoretical computer science, University of lübeck.

[4] Konstantin Skodinis. Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time, Journal of Algorithms, Volume 47, Issue 1, April 2003.

[5] John D. Hobby. Drawing graphs with MetaPost, AT&T Bell Laboratories, 1992.

[6] John Morris. Algorithm Animation: Using algorithm code to drive an animation. ACE '05 Proceedings of the 7th Australasian conference on Computing education - Volume 42, 2005.

[7] John D. Hobby. Drawing Boxes with MetaPost. [Online].
Available: http://www.tug.org/docs/metapost/mpboxes.pdf

[8] Yifan Hu. Efficient and High-Quality Force-Directed Graph Drawing, Wolfram Research Inc., USA.

[9] Stephen G Kobourov. Spring Embedders and Force-Directed Graph Drawing Algorithms, 2012.

[10] Brian Beckman. Theory of Spectral Graph Layout, Tech. Report MSR-TR-94-04, Microsoft Research Force directed graph drawing.

[11] Dennis Hotson. Springy.js – A force directed graph layout algorithm in JavaScript. [Online]
Available: http://getspringy.com/