Animating Algorithms: Project Specification

Steven Cooper (sjc209), Graham Le Page (gpl26),
Zhan Li (zrl12), Robert McQueen (ram48),
Andrew Medworth (am502), Neofytos Mylona (nm314),
Sidath Senanayake (sas58), Alan Treanor (ajit2)

29th January 2004

Contents

1 Introduction

This is the specification document for the "Animating Algorithms" (Topic 8.5) Computer Science Tripos Part IB / Part II (General) group project. The authors are the members of group "Alpha", listed individually above[1]. This project is being overseen by Steven Hand (`smh22@cam.ac.uk`).

2 Background

The field of algorithms and data structures is one of the hardest parts of Computer Science to teach. When demonstrated on blackboards or overhead projectors, diagrams can get extremely messy and hard to follow; even when the steps of an algorithm are clearly displayed on paper, it is not always immediately clear what operations have been done between each step.

Computer animation would be an ideal way to demonstrate algorithms: the actions taken by the computer at each stage could be demonstrated at whatever level of abstraction is most helpful. Users could move through an algorithm's execution at their own speed, going especially slowly over difficult steps. Critical information could be shown in a clear, intuitive way (for example using colour coding), and this could be reinforced with clean, smooth graphical animations showing exactly what is going on internally as the algorithm is executed. The cost of the algorithm in time and space on different sizes of input could also be displayed.

Clearly, a system to animate any conceivable algorithm working on any imaginable data structure is well beyond the scope of a six-week project, but we believe it should be possible to provide an extensible system with animation primitives enabling the illustration of algorithms which operate on vectors (for example sorting, and searching algorithms) or graphs (for example shortest-path and strongly-connected component finding algorithms) within the available time.

3 Assumptions

- The "animation script" and the algorithms themselves will be written in Java. We will not implement our own special language for describing algorithms; instead, we will implement and document a Java API which permits new algorithms to be animated.
- The user will be able to specify initial data in the following ways:
  - For *vectors*: the size of the initial array (up to a maximum of 20 elements), and the initial contents of each cell (which must be `ints`: there is no need to demonstrate algorithms on multiple data types). There will be no option to invert the ordering, as this will not aid understanding of the algorithm.
  - For *graphs*: the ability to specify the number of nodes (up to a maximum of 10), and input the elements of the connectivity matrix (which must all be integers).

[1]To obtain group members' email addresses, simply append "`@cam.ac.uk`" to the usernames in brackets after each person's name.

- The user will set the input data once, before the animation starts, and after that time he/she will not be able to alter the data without starting the algorithm again.

4 Facilities to be provided

There are some facilities that the system will be useless without, and whose implementation is therefore essential. There are others, however, which are not crucial but will usefully extend the functionality of the system. These facilities will be considered optional and may or may not be implemented, subject to time constraints.

Essential facilities

- Support for algorithms on vectors and graphs
- Clear and smooth graphical animation
- Text explanation of each step as it happens
- Ability to step through algorithms, both forwards and backwards, at the click of a mouse
- Ability to specify the algorithm's initial data or input (plus the ability to generate random initial data)
- Ability to add new algorithms using existing animation primitives (by writing new Java classes using an existing API)
- Well-documented API for creation of new animation primitives and data structures, potentially allowing the implementation of different types of algorithm from the ones done here (e.g. heap management routines)
- Cross-platform operation using Java
- User-friendly interface
- In order to demonstrate the system, we will initially implement three sorting algorithms and two graph algorithms:
  - insertion sort,
  - Quicksort,
  - radix sort,
  - Dijkstra's algorithm (for finding the shortest path between two nodes in a graph), and
  - Kruskal's algorithm (for finding the minimum spanning tree of a set of nodes in a graph).

Optional facilities

- Ability to vary the animation speed (between two reasonable values)
- Display of the time and space costs of the algorithm (in text or graphical mode)
- A graph editor which allows the user to place the nodes on the animation canvas, and to specify the edges connecting the nodes using mouse clicks
- Support for bipartite graphs
- Implementations of more algorithms, e.g. Prim's algorithm, shell sort, strongly connected component finding algorithm

5 Environment

- The system will be written in Java 2 using the Eclipse IDE.
- The GUI will be implemented in the Swing windowing toolkit.
- Version control will be provided by `cvs`: the repository will be located on `ned.ucam.org`, a student-operated server located in Corpus Christi College.

6 Overall system architecture

The main functionality of the system can be broken down into four areas: algorithms, animators, queues, and shell. Each will be represented as a package in Java. The following sections contain a brief description of each package, the classes within, and their interactions.

6.1 Algorithms

The classes within this package will ultimately implement the algorithms which the system is to animate: they will contain the instructions to the other parts of the program for animating the steps that are taken by the algorithm, provide the narrative for explaining to the user what is happening, and any other information which is displayed.

The class hierarchy for this package can be seen in Figure 1. The abstract class `Algorithm` is the parent class for all of the classes within this package, and an abstract class is descended from it for each group of algorithms which are being implemented. We will implement two types of algorithm, `GraphAlgorithm` and `VectorAlgorithm`. These abstract classes will specify methods for querying information about the algorithm, such as its name and description, as well as setting the initial data for the algorithm and instructing it to execute.

6.2 Animators

The UML class diagram for this package can be seen in Figure 2. It contains a set of abstract classes descended from `Animator` which have methods and inner classes to specify the animation primitives available to each group of algorithms. There will initially be two classes, `GraphAnimator` and `VectorAnimator`, which will specify the methods necessary for animating graph and vector algorithms.

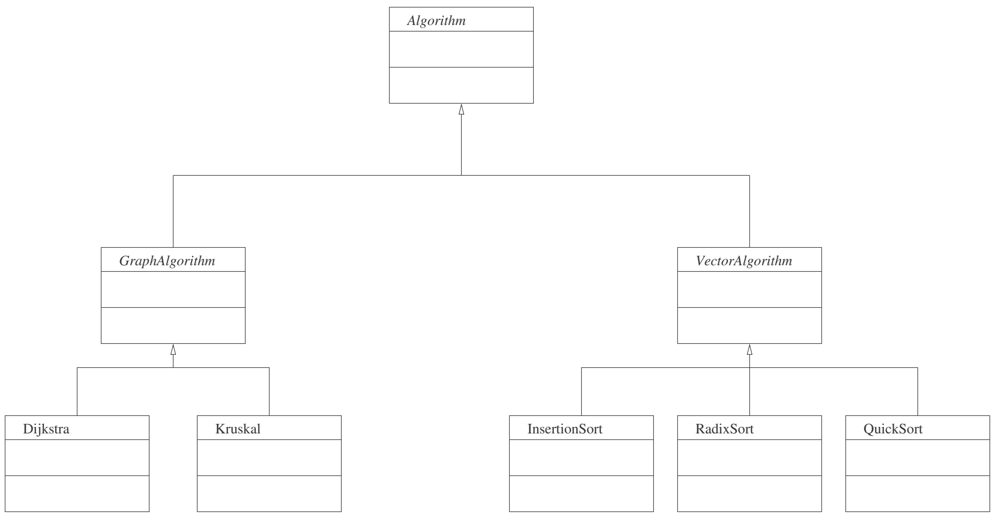

Figure 1: UML class diagram for algorithm package

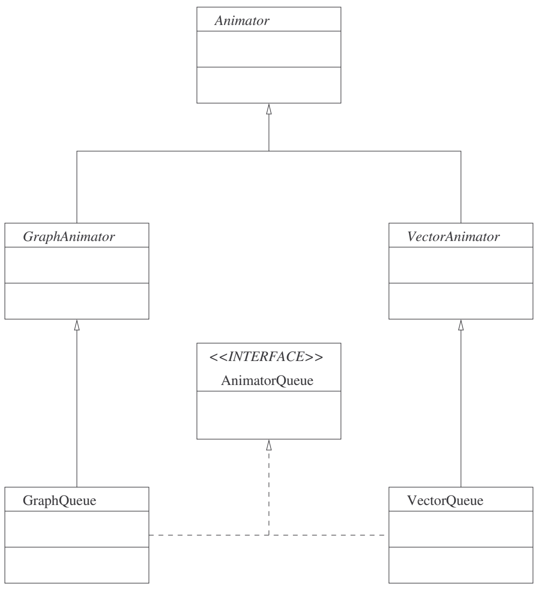

Figure 2: UML class diagram for animator and queue packages

6.3 Queues

The class hierarchy for this package can also be seen in Figure 2. It contains an interface `AnimatorQueue`, initially implemented by two classes `GraphQueue` and `VectorQueue`, which extend their respective `Animator` classes (from the animators package). The queues form a layer of abstraction between the algorithms and the actual implementation of the animators, and allow the algorithms to run to completion before any animation is shown to the user. When the algorithm uses an animator primitive, the queue internally records the requested action, and returns control to the algorithm.

Every so often the algorithm will call a special animation primitive which signals a *checkpoint.* These checkpoints are the junctures at which the animation will pause and the user will have the opportunity to go forward or back — essentially, each sequence of animation primitives between checkpoints constitutes one step of the algorithm. For example, if we were implementing insertion sort, one step might consist of the insertion of a single value into the vector: we might want to move a pointer along the vector until we find the correct place for the value, move all the other data items up to make room, insert the value into the gap, and update a pointer showing what portion of the vector is sorted. Clicking the "next step" button would then animate the insertion of the next item.

As the algorithm progresses, the queue builds up a list of all of the animator primitives which the algorithm used to get between one checkpointed state of the animator and the next. The shell can then provide the queue with an actual implementation of the animator primitives, and use the `AnimatorQueue` interface to instruct the queue to execute each batch of queued primitives in sequence on the actual animator.

The more important functionality of this package is that, at each checkpoint reached when playing back the stored events, it stores the state of the animation canvas and the data structures of the animator. This will allow the user to roll the animator back to previous states so the user can revisit previous steps of the algorithm, and the queue retains the steps that need to be done to return the animator to subsequent states. There will be methods in the `AnimatorQueue` interface available to the shell which will cause the animation to move forwards and backwards between checkpoints. To move backwards, the state of the canvas and data structures at the previous checkpoint will be restored, and the queue's internal structures will be updated accordingly. To move forwards, the next animation primitives will be executed in sequence until the next checkpoint is reached.

It should be noted that it would not be possible to skip forwards to the next checkpoint (without animation) unless the user has gone backwards and then forwards again, because the state of the animation canvas would not exist yet. This functionality could be simulated by running the animation very rapidly when the user wants to skip forwards to the next checkpoint state.

6.4 Shell

The shell package contains classes which provide the user interface and all of the visible elements of the system (such as the actual animation canvas). The concrete implementations of the `GraphAnimator` and `VectorAnimator` abstract classes are here, along with a `Shell` class which contains the entry point for the whole system and controls the main program flow.

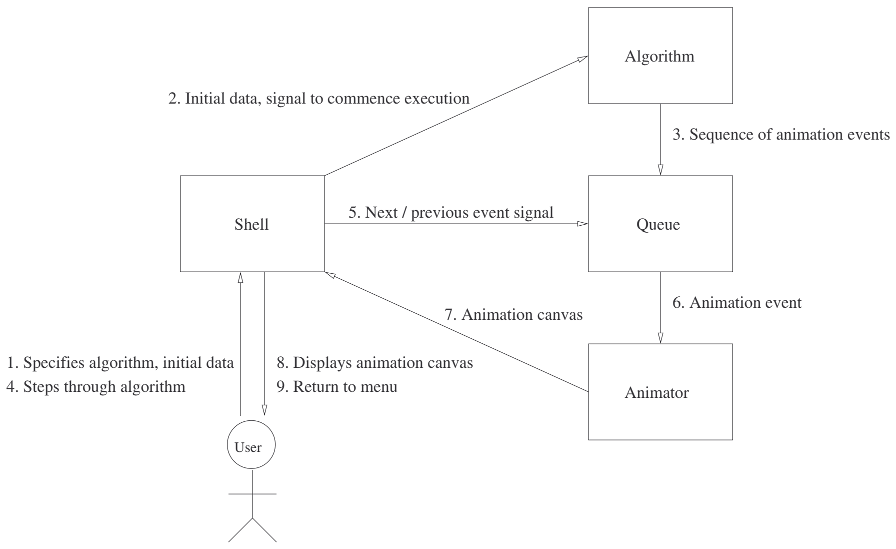
Algorithm
2. Initial data, signal to commence execution
3. Sequence of animation events
Shell
5. Next / previous event signal
Queue
6. Animation event
7. Animation canvas
1. Specifies algorithm, initial data
4. Steps through algorithm
8. Displays animation canvas
9. Return to menu
Animator
User

Figure 3: Diagram showing data flow between system components

The shell can investigate the algorithm package (via introspection) to see which classes are available, and present a list of algorithms for the user to choose from. When an algorithm is selected, the shell will allow the user to input the initial data for the algorithm, or generate some at random. The shell then creates an instance of the appropriate animator, wraps this in the appropriate queue class, and passes this to the algorithm, which is then run to completion. The shell then displays the animator on screen, steps through the algorithm's stages as the user requests them (using the `AnimatorQueue` interface), shows any explanatory text provided by the algorithm, and shows which step is active. The shell will either advance the animation at a preset speed, or the user can step forwards and backwards through the stages manually. After an animation has run to completion, the user can return to the initial screen and choose a new algorithm or exit.

To clarify the way data moves around the system (and how control is transferred), see Figure 3.

7 Overall inputs, functions, and outputs

User inputs

- Algorithm selection (from the ones implemented)
- Initial data (keyboard input for vector elements, graph edge costs) including ability to generate random data and special cases (such as already sorted data)
- Navigation through steps of algorithm using buttons
- Animation speed *[optional]*
- Graph node locations and other layout information (using mouse, in graph editor) *[optional]*

Functions

- Simulate algorithm using given initial data, storing the state at each stage to facilitate forward and backward movement, plus information displays such as time and space cost
- Animate each step in the algorithm when the user moves forward

Outputs

- Animation displayed on screen
- Textual explanation of steps
- Time and space cost display *[optional]*

8 Specification of components and test requirements

This section has a similar structure to the one on overall system architecture. We will deal with system components grouped by package. For each class (or type of class) we will describe its purpose and intended function, its inputs and outputs, its main public methods, and we will suggest a possible test strategy. The lists of public methods are not exhaustive and may be subject to change: these sections are designed by way of an alternative to abstract classes.

8.1 Algorithms

Purpose

Each concrete algorithm class (inheriting from `GraphAlgorithm` or `VectorAlgorithm` as appropriate) implements a certain algorithm which can be displayed to the user, as well as including the code to animate and explain this algorithm to the user as it executes.

Inputs

The algorithm receives its initial data, and an instance of an object which extends the `Animator` appropriate to the class of algorithm. For example, a sort algorithm could receive a vector of integers, and an instance of a `VectorAnimator` class.

Outputs

When executed, the algorithm makes a sequence of method calls on the provided object which extends its appropriate `Animator` abstract class. These method calls represent the sequence of steps necessary to present an explanation to the user of the execution of the algorithm.

Public methods in abstract `Algorithm` class

- *static* `String getName()`
- *static* `String getDescription()`
- `void setAnimator(Animator)`
- `void setData(...)`
- `void execute()`

Testing

Each algorithm can be tested independently of the main system by making a test program that provides a `TestAnimator` class, which rather than queueing or animating, implements stub functions to merely display the sequence of method calls which were made by the algorithm. The program can then run the algorithm on a variety of possible inputs to ensure that the correct results are generated, even in extreme cases such as a vector full of `0`s.

8.2 Animators

Purpose

The animator abstract classes define all of the methods which are available for the algorithms to call. All of the necessary animation primitives are included in these abstract classes, which are implemented by classes in the queue and shell packages, for queueing and actually displaying the animation events respectively. The initial implementation will include two `Animator` subclasses for graph and vector animation primitives. Actual drawing implementations of these abstract classes must support the ability to save their state to an opaque `Checkpoint` object which can be later used to restore the animator to a previous state, using the memento design pattern.

Public methods in abstract `Animator` class

- `void setSteps(String[])` — define a series of strings to explain what the algorithm does
- `void setCurrentStep(int)` — highlight currently active step
- `void showMessage(String)` — show text to user to explain current actions
- `Checkpoint saveState()` — save the state of the animator so this stage of the algorithm can be returned to later when required
- `void restoreState(Checkpoint)` — return the animator to the state it was in when a given checkpoint was taken

Public methods to be provided by abstract `GraphAnimator` class

Animation primitives on the overall graph:

- Create graph (specify nodes and paths with costs and labels)

Animation primitives on a node in the graph:

- Change node label
- Highlight node (to indicate current focus of algorithm)
- Shade node (to indicate membership of a set of nodes or paths)

Animation primitives on a graph edge (join between two nodes)

- Change edge label
- Highlight edge (to indicate current focus of algorithm)
- Shade edge (to indicate membership of a set of nodes or paths)

Public methods to be provided by abstract `VectorAnimator` class

Animation primitives on a vector (array of `ints`):

- Create vector (with initial data and label)
- Change vector label
- Change vector element
- Swap elements at two pointers
- Copy element from one pointer to another
- Compare two elements (uses pointers to indicate which are being compared)
- Delete element at pointer
- Insert element at marker (essentially just several "simultaneous" copies followed by an element alteration)
- Split vector into two at a marker (for Quicksort)
- Highlight columns of the vector (for Radix sort)

Animation primitives on a *marker* (an arrow pointing at the boundary between two entries in the vector):

- Create marker (with label and position)
- Delete marker
- Change marker label
- Move marker

- Highlight marker (to indicate current focus of algorithm)

Animation primitives on a *pointer* (an arrow pointing at a specific entry in the vector):

- Create pointer (with label and position)
- Delete pointer
- Change pointer label
- Move pointer
- Highlight pointer (to indicate current focus of algorithm)

8.3 Queues

Purpose

The basic purpose of these classes was described in Section 6.3. The queues act as a layer of abstraction between the algorithms and the animators that are implemented in the shell, and implement a journal-like functionality which delays animation primitives until the shell is ready for them to be displayed, allows the animator to be rolled back to former states, and events replayed to return to later states. By extending the same abstract parent classes as the actual animators, the overhead of the queueing functionality is hidden from the authors of the algorithms, who can use the imperative-style animation primitive methods, but the implementation is rendered simple because ultimately the queue needs only to store and replay similar method calls in sequence.

Because the queue classes are descended from the abstract animator classes such as `GraphAnimator` and `VectorAnimator`, each of these needs a counterpart queue class, namely `GraphQueue` and `VectorQueue`. To simplify implementation of the queues, and avoid code duplication, a common class will be created for the queue back-end, to manage the storing of checkpoint objects, and queues of objects to represent the intermediate events. The queue classes can use the adapter design pattern, and implement their functionality with this common class. The common class can handle calls to the `Animator` class easily, and can use introspection to call the animation primitive methods on the actual animator. In this way, the code can be kept generic, and the implementation of queues for new classes of animator kept to a small amount of code.

Inputs

A series of method calls from an implementation or an `Algorithm` class, representing a sequence of changes to be made to the animation canvas between stages of the algorithm. User input via the shell (e.g. forward and back instructions).

Outputs

The same series of method calls as was provided as input, at the control of the `AnimatorQueue` interface, restoring past checkpoints to go back, and re-calling method calls to go forwards.

Public methods in `AnimatorQueue` interface

- `void nextCheckpoint()`
- `void prevCheckpoint()`
- `boolean isNextCheckpoint()`
- `boolean isPrevCheckpoint()`

Testing

Testing of the queue classes requires the existence of a working test algorithm which generates a full variety of animation primitive method calls, and a test animator which displays which method calls are made. A simple test program can then be made to move forwards and backwards through the queued steps, and ensure that the output of the test animator matches the expected changes.

8.4 Shell

Purpose

The shell is a collection of classes that contain all of the code to implement the interface that the user interacts with. The shell allows the user to choose an algorithm and input or generate data, and then uses classes from the other packages which run the relevant algorithm and queue the animation primitives. The shell also contains the classes to render output to the display, which are extended from the abstract subclasses of `Animator`. These animators will implement an internal interface which allows the shell to control playback speed, pausing, etc, independently of which type of animation is being implemented. The shell will instantiate all of the objects as required, such as the algorithm, queue and actual animator classes, will pass data between components, such as the algorithm's initial data, and control the queue and animator objects to e.g. move between steps, and pause or adjust animation speed. Where possible, the code that distinguishes between specific types of algorithm (ie those requiring a `GraphQueue` and a `GraphAnimator` implementation) will be kept to a minimum, behaving as an abstract factory so that the shell can work for new algorithms and types of algorithm without modification.

Inputs

The shell will accept, from the user, the algorithm choice and the provision of initial data for the algorithm. When the animation has been prepared and being presented, a series of animation playback options such as previous, next, play and pause will be provided to the user. The shell can also make use of Java introspection to enumerate the available algorithms within the algorithm package, and use this to present the list to the user.

Outputs

As a graphical component, the shell produces all of its output on screen, but this can be broken down into two categories. One is interaction whilst the user selects an algorithm and enters data, and the other is the actual display of the animation (and associated information) on screen.

Public methods in `Shell` class

- *public static* `void main(String[])` — the entry point of the entire system

Testing

Testing is to be broken down into two sections to mirror the two functions of the shell. For algorithm selection and data entry, the functionality of the relevant Swing components will be tested with algorithms containing stub functions to verify that the right method calls are being made. In addition, the shell's ability to cope with invalid user data (such as entering a string into a vector of numbers) will be tested. To test the animation section, further test algorithms will be created which call methods to test all of the animation primitives and combinations thereof, to ensure the display is correctly updated and that user interaction or interruption during animations (such as clicking previous, next or pause) is handled appropriately.

9 Final acceptance criteria

- The system must allow a user to pick an algorithm from a selection of at least five, where there must be at least two different types of algorithm;
- the system must allow a user to input data to be used with the selected algorithm;
- the system must compute each step of the algorithm correctly using the data it receives;
- the system must animate and explain each step of the algorithm on the screen, allowing the user to review each step individually and move forwards and backwards at will;
- all other features on the "essential" list must be implemented;
- the user documentation must explain how to use the system as it is currently implemented;
- the developer documentation must explain how to use the APIs provided to extend the system with new algorithms and animations.

10 Division of tasks

The division of tasks is described in Table 1. Group members with *(backup)* written next to their names will be called upon to perform that task in the event that the others cannot complete it on their own, or extra effort is needed.[2]

[2]Document word count: 3939

Manager	Graham Le Page
Librarian	Robert McQueen
Secretary	Andrew Medworth
User documentation	Neofytos Mylona Graham Le Page Alan Treanor Andrew Medworth *(backup)*
Design & Coding	Robert McQueen Steven Cooper Andrew Medworth Sidath Senanayake Zhan Li *(backup)*
Troubleshooting & Testing	Zhan Li Alan Treanor Sidath Senanayake *(backup)* *As many others as are required*

Table 1: Division of project tasks