University of Glasgow | School of Computing Science

Algorithm Animator

Andrei-Mihai Nicolae

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

Level 4 Project — March 22, 2017

Abstract

Understanding algorithms is both very common and hard for developers in general, regardless of their level of expertise. Even the fundamental ones, such as Dijkstra's algorithm for finding the shortest path between two nodes in a graph, are quite complicated to grasp. Many studies show that visualizing an algorithm and its steps make understanding it much easier. In this report, we will present an Algorithm Animator built specifically for solving this problem in a modern, responsive and efficient manner. Among others, we will also show why certain design decisions (e.g. making it a native desktop app instead of a basic jar, using material design for the user interface), the implementation choices and the evaluation results make this tool a viable option for software engineers when it comes to learning different kinds of algorithms.

Education Use Consent

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

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

The travel ling salesman problem, or TSP as it is commonly known, is a combinatorial optimisation problem where, given a number of cities and distances from any one city to any other city, the goal is to find the shortest tour that visits all cities only once. Being an NP-complete problem, no algorithm is known that is able to find the optimal solution within a feasible amount of time. The lack of an algorithm for finding the optimal solution has resulted in many alternative heuristics and algorithms being described for providing a good approximation of the optimal solution. In an educational setting, trying to introduce the concepts of some of these algorithms to students that may have never encountered such algorithms before is not always a straightforward task.

1.1 Aims

The aim of this project is to visualise a variety of algorithms operating upon TSP instances, creating an application for use as a t eaching aid by a lecturer to help students understand how each algorithm develops a solution. The TSP has certain properties that make it a good candidate for attempting to visualise the algorithms operating upon it. The problem itself is relatively simple to represent visually by plotting the coordinates of the cities to be visited, with edges connecting the cities to represent sections of a tour. One way of trying to increase the ability of students to understand new algorithms is by providing them with visual representations of steps or features of the algorithms. While still images could be used as visualisations, this is often not an optimal way of helping others to understand the execution of an algorithm. Providing animations better allows for understanding of how algorithms progress towards a solution, resulting in a more focused aim of this project being animations of algorithms operating upon TSP instances for teaching purposes.

1.2 Motivation

With the TSP being a popular problem, a large number of sample instances and optimal solutions exist in the form of the TSPLIB library [1], which is a good source for being able to evaluate solutions generated by the algorithms against optimal solutions. The amount of research into the TSP as a combinatorial problem has even spawned a book devoted to the subject [3], the authors of which also developed an application known as Concorde [2]. Concorde is a program that is able to produce optimal solutions to TSP instances, solving 106 of the 110 TSPLIB instances, the largest of which is a tour through 15,112 German cities.

Attempts to visualise algorithms for educational purposes have been undertaken many times before, such as the algorithm toolkit developed by John Morris [4]. This toolkit was essentially a framework into which

new algorithms could be implemented, alongside visualisations, aiding students in their understanding of the algorithms. This previous evidence of visualisations being successfully developed for learning various algorithms suggests that using a similar approach to the visualisation of algorithms upon the TSP could result in a useful application covering a comprehensive range of algorithms.

1.3 Contributions

1.4 Report Content

The rest of the report will analyze the background of animators and why they were proven useful, as well as cover all the steps in gathering requirements, designing, implementing, testing and evaluating the tool.

- Chapter 2 covers work related to the purpose of algorithm animators and why they are useful
- Chapter 3 goes into how the problem was analyzed and what requirements were gathered through project meetings and discussions with Algorithmics students.
- Chapter 5 explains the design decisions behind the tool and illustrates various lessons learned and problems faced along the way.
- Chapter 6 goes into the implementation details of the animator.
- Chapter 7 show how extensive unit, integration and other types of testing (e.g. smoke, end-to-end) were undergone and why they were essential to the development of the application.
- Chapter 8 details the overall results of the project.

Chapter 2

Background

2.1 Related Work

Chapter 3

Requirements

3.1 Problem Analysis

3.2 Requirements Gathering

3.3 Functional Requirements

3.4 Non-Functional Requirements

Chapter 4

Planning

4.1 Agile

4.1.1 Kanban Board

4.1.2 Issues & Bug Tracking

Chapter 5

Design

5.1 Architecture

5.1.1 EDA (Event-driven Architecture)

5.2 Native Desktop App vs. Jar

5.3 Electron

5.4 Vis.js

5.5 Material Design

Chapter 6

Implementation

6.1 High-Level Overview

The implementation of the system is based heavily upon a Model-View-Controller architecture. The model portion of the system provides a collection of algorithms with methods allowing them to be run, giving the ability to integrate the algorithms into a range of front-ends. This allows the user interface to be implemented independently of the algorithms, enabling a new interface to be placed on top of the algorithms and negating the need to re-implement any of the underlying algorithms specifically for a new user interface. Java's Swing GUI classes provide both the view and controller aspects of the architecture. The visualisation classes provide the view, while the control object classes, such as `JButton`, combined with `ActionListener` classes provide the control over the application. Although Swing is used for the front-end implementation in this project, the nature of the MVC architecture means that an interface developed in any language could be used, as long as it is able to communicate with the Java methods defined by the algorithms.

6.1.1 Problem Instance Representation

To represent the problem instances, a `TSPInstance` class is used. This class represents a fully connected graph of `AbstractCity` objects, where each city has a unique integer identifier, starting from 0, and can calculate the distance from itself to every other city in the instance. As several different distance calculations are used within the application, such as Eulerian distances, concrete classes extending `AbstractCity` must implement their own distance calculation function.

While the distance calculations could be performed on-the-fly, this gives a high constant-time cost for executing the large amount of distance calculations required by every algorithm. Since distances are represented as floating-point values by the application, which are approximations of values, there is the chance that on-the-fly calculations could yield differing distances between the same two cities. In the application, a distance matrix is used to store the distances between any two given cities, where the rows of the matrix are distributed out to the subclasses of `AbstractCity` so that each city is able to know the distance from itself to every other city. The use of a distance matrix ensures that there is a low constant-time cost for distance lookups, and that subsequent distance lookups will always have yield the same value.

Chapter 7

Testing

7.1 High-Level Overview

The implementation of the system is based heavily upon a Model-View-Controller architecture. The model portion of the system provides a collection of algorithms with methods allowing them to be run, giving the ability to integrate the algorithms into a range of front-ends. This allows the user interface to be implemented independently of the algorithms, enabling a new interface to be placed on top of the algorithms and negating the need to re-implement any of the underlying algorithms specifically for a new user interface. Java's Swing GUI classes provide both the view and controller aspects of the architecture. The visualisation classes provide the view, while the control object classes, such as `JButton`, combined with `ActionListener` classes provide the control over the application. Although Swing is used for the front-end implementation in this project, the nature of the MVC architecture means that an interface developed in any language could be used, as long as it is able to communicate with the Java methods defined by the algorithms.

7.1.1 Problem Instance Representation

To represent the problem instances, a `TSPInstance` class is used. This class represents a fully connected graph of `AbstractCity` objects, where each city has a unique integer identifier, starting from 0, and can calculate the distance from itself to every other city in the instance. As several different distance calculations are used within the application, such as Eulerian distances, concrete classes extending `AbstractCity` must implement their own distance calculation function.

While the distance calculations could be performed on-the-fly, this gives a high constant-time cost for executing the large amount of distance calculations required by every algorithm. Since distances are represented as floating-point values by the application, which are approximations of values, there is the chance that on-the-fly calculations could yield differing distances between the same two cities. In the application, a distance matrix is used to store the distances between any two given cities, where the rows of the matrix are distributed out to the subclasses of `AbstractCity` so that each city is able to know the distance from itself to every other city. The use of a distance matrix ensures that there is a low constant-time cost for distance lookups, and that subsequent distance lookups will always have yield the same value.

Chapter 8

Conclusions

8.1 High-Level Overview

The implementation of the system is based heavily upon a Model-View-Controller architecture. The model portion of the system provides a collection of algorithms with methods allowing them to be run, giving the ability to integrate the algorithms into a range of front-ends. This allows the user interface to be implemented independently of the algorithms, enabling a new interface to be placed on top of the algorithms and negating the need to re-implement any of the underlying algorithms specifically for a new user interface. Java's Swing GUI classes provide both the view and controller aspects of the architecture. The visualisation classes provide the view, while the control object classes, such as `JButton`, combined with `ActionListener` classes provide the control over the application. Although Swing is used for the front-end implementation in this project, the nature of the MVC architecture means that an interface developed in any language could be used, as long as it is able to communicate with the Java methods defined by the algorithms.

8.1.1 Problem Instance Representation

To represent the problem instances, a `TSPInstance` class is used. This class represents a fully connected graph of `AbstractCity` objects, where each city has a unique integer identifier, starting from 0, and can calculate the distance from itself to every other city in the instance. As several different distance calculations are used within the application, such as Eulerian distances, concrete classes extending `AbstractCity` must implement their own distance calculation function.

While the distance calculations could be performed on-the-fly, this gives a high constant-time cost for executing the large amount of distance calculations required by every algorithm. Since distances are represented as floating-point values by the application, which are approximations of values, there is the chance that on-the-fly calculations could yield differing distances between the same two cities. In the application, a distance matrix is used to store the distances between any two given cities, where the rows of the matrix are distributed out to the subclasses of `AbstractCity` so that each city is able to know the distance from itself to every other city. The use of a distance matrix ensures that there is a low constant-time cost for distance lookups, and that subsequent distance lookups will always have yield the same value.

Bibliography

[1] TSPLIB website. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/, October 2009.

[2] Concorde Application Home Page. http://www.tsp.gatech.edu/concorde.html, October 2010.

[3] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Travelling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, USA, 2006.

[4] John Morris. Algorithm Animation: Using Algorithm Code to Drive an Animation. In *ACE '05: Proceedings of the 7th Australasian conference on Computing education*, pages 15–20, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.