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

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 Background

With the TSP being a popular problem, a large number of sample instances and optimal solutions exist in the form of the TSPLIB library [**?**], 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 [**?**], the authors of which also developed an application known as Concorde [**?**]. 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 [**?**]. 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 Motivation

The major motivation of this project is in providing a tool to make teaching of a set of algorithms easier for both a lecturer to present and for the students to understand. This provides the benefit of giving students something that they may interact with in order to further increase and reinforce their knowledge of the algorithms. The TSP allows easy visualisation itself in its most basic state by drawing a tour. This alone suggests a good possibility for being able to animate a great number of algorithms operating upon it, providing a useful tool for teaching a large range of algorithms.

1.4 Report Content

The remainder of this report will discuss how producing a solution to the problem of visualising algorithms was undertaken.

- Chapter 2 covers the detailed requirements of the project.
- Chapter 3 explains the design and implementation details of the project.
- Chapter 4 details the overall results of the project.

Chapter 2

Requirements

2.1 Problem Analysis

To derive requirements for the application, requirements gathering was undertaken during project meetings to try and define the basic expectations of the application.

The objective appeared to be in how to visualise algorithms that may take a large number of iterations to complete, showing them execute until completion in a suitable time. However, the problem is that while doing this, the application must provide informative data in the animation which could be easily picked up on by a viewer, such as a student, with little understanding of the algorithm, helping illustrate to them what the algorithm is doing.

The basic visualisation requirements given for these algorithms were that the current tour should be visible along with its cost and that the application should support altering of input parameters to see how modifying parameters changes the performance of algorithms. Algorithm-specific features were to be identified and visualised in an appropriate manner that provided a representation of the algorithm which students could benefit from as a learning aid.

2.2 Proposed Solution

The proposed solution was to develop an application able to display animations of algorithms using the Java programming language. As Java provides both a large set of class libraries, including extensive graphical user interface libraries which support hardware rendering, this made it a perfect candidate for implementing a solution to the problem at hand. Java also has

Chapter 3

Design and Implementation

3.1 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.

3.1.1 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.

3.1.2 Furthest Insertion

Origins

Furthest insertion, proposed in [**?**], attempts to overcome the naïvety of the nearest neighbour algorithm somewhat by trying to consider the solution as a whole in its construction. It constructs a tour upon the basis of accepting what may potentially be the worst moves first, getting rid of the costly paths early on and setting up a good structure for the overall solution to fit into, leaving the less costly paths for inclusion at a later point in the algorithm.

Furthest insertion is used to quickly construct cheap tours for the TSP, often for passing into local search algorithms for further improvement.

Description

Furthest insertion begins by selecting the two cities that are furthest apart in the graph and joining these two cities in a tour. Following this, the distances from all unvisited cities in the graph to the cities currently on the tour are calculated, with the intention of finding the unvisited city whose minimum distance to a city on the tour is maximal. This unvisited city is then inserted in the tour in the position that will cause the minimum overall increase in the length of the tour. This process of selecting and incorporating cities is then repeated until a tour over all the cities in the graph is constructed.

Implementation

Furthest insertion in this project is implemented as described above.

Visualisation

To visualise the furthest insertion algorithm the tour under construction is updated as each city is added to it. This allows a viewer to see how the tour develops as new cities are incorporated into its structure.

Discussion

Although furthest insertion constructs a reasonable tour which is generally much shorter than a heuristic such as nearest neighbour will produce, it is still not immune to producing a tour with costly crossovers of edges.

3.2 Local Search Algorithms

Local search is a metaheuristic that can be used to iteratively improve upon an existing solution to an optimisation problem via minimising some evaluation function. In the case of the TSP the goal is to minimise the value of the distance required to visit all cities on a tour and return to the initial city.

Local search is utilised in solving problems that can have multiple possible solutions, not necessarily all optimal. In order to operate, local search must be able to generate new solutions, which it can then apply its evaluation function to so as to determine the viability of each solution as a potential next solution to move to.

A move generator is used to create a neighbourhood of solutions, via some modification to the current tour. Each of the solutions in this neighbourhood must have the cost function applied to them in turn, enabling the local search algorithm to move to the next solution state which is most favourable to the cost function.

Local search will then proceed in repeatedly generating neighbourhoods of solutions, evaluating them, and moving to the most favourable until some termination condition is reached. Termination conditions can be dependent upon bounds on iterations of the local search algorithm, time bounds, when a certain goal — such as the cost of a tour — has been reached or surpassed, or even when the local search hits a local minima and is unable to find any improving neighbours in its move generation phase.

The wide range of local search methods and heuristics in existence has caused entire publications to be produced on the subject of local search alone [**?**].

To generate a neighbourhood of solutions using this edge exchange method, all pairs of 2-opts within the tour must be generated and evaluated. This requires a neighbourhood of worst-case size $O(n^2)$ for a tour of size n.

When evaluating new tours generated via 2-opts, the cost of performing the actual 2-opt can take up to $O(n)$ time (in an array representation of the tour), followed by a cost calculation iterating over all edges in the tour, also bound by $O(n)$. However, a method requiring $O(1)$ time to evaluate the cost of a 2-opt on a tour can be used. We need only know two cities, i and j, which are the ends of the path to be reversed by a 2-opt operation. In a symmetric TSP instance the reversed path will yield the same cost as the non-reversed path, so we need only calculate the cost of breaking the two existing edges and rejoining the paths with new edges. If i_{prev} and j_{next} are the predecessor and successor of i and j, respectively, in the tour, we need only calculate the cost of breaking $dist(i_{prev}, i)$ and $dist(j, j_{next})$, and rejoining with $dist(i_{prev}, j)$ and $dist(i, j_{next})$.

2-opts are not the only way of generating neighbouring solutions. A range of other optimising operations exist, from 3-opts, involving the interchanging of 3 edges, to k-opts involving the interchanging of k edges. A highly-optimised exchange heuristic is the Lin-Kernighan k-opt heuristic, which dynamically selects the number of edges to be interchanged in any optimising move [**?**].

Chapter 4

Conclusion

The aim of this project was to create an application that could be used as a teaching aid for introducing students to new algorithms, focusing on local search algorithms in particular. An application was produced that provides animations for a range of algorithms, with an evaluation of the animations conducted by participants who were believed to have some understanding of the algorithms.

The results of the evaluation suggest that, overall, the project was successful in helping enhance or reinforce an understanding of the algorithms through the use of animations and that the application would be suitable for use in a teaching situation. This satisfies the main aim of the project in producing an teaching tool to aid in understanding of the algorithms. The application that was produced should enable students learning the algorithms for the first time to build up a good understanding through visualisations of a comprehensive collection of algorithms from the 3 classes of algorithms implemented.