Some Title

Sean Hunter Brooks
seanhunterbrooks@gmail.com
Git: @astronomerhunter

September 29, 2017

Abstract

Imagine you're going on a road trip and you want to visit some number of cities while spending the shortest amount of time on the road. Obviously some paths are more efficient than others; you'd want to avoid routes like New York to Los Angles to Boston and instead favor maybe NYC to Boston to LA. How does one find the most efficient path? Would you be willing to settle for a path that takes only slightly longer than the most efficient path? Investigation of these questions is an interesting

1 Introduction

Imagine you're going on a road trip and you want to visit some number of cities while spending the shortest amount of time on the road. Obviously some paths are more efficent than others; you'd want to avoid routes like New York to Los Angles to Boston and instead favor maybe NYC to Boston to LA. How does one find the most efficent path? Would you be willing to settle for a path that takes only slightly longer than the most efficent path? These questions are points of whats called the Traveling Salesmen Problem, or TSP.

The generalization of the TSP is previlent in the world today. Package delivery, circut board manufactoring, and product procurement in warehouses all involve our generalized path finding problem. At the moment, mankind has only developed one reliable method to solve this general problem, but the drawbacks of the surefire method are extreme. Because of this, many techniques have been developed to estimate the optimal route. These estimations can come very close to the optimal solution with relatively low reprocussions of getting there. The software in this Github repository offers the investigative reader the ability to learn about various predictive techniques.

This Github repository: - walks the reader through the details of the problem at hand - provides sample algorithms that can be used to solve the problem - allows the user to create custom algorithms to solve the problem - creations visualizations of the problem and solution

Ultimately, this software and accompaning documentation was created with the following concepts in mind: - Readability. Anyone with a brief introduction to Python should be able to interpret the code and with a college background in Mathematics one should be able to digest the documentation. - Education. Complex concepts are assemblies of simplier ones; learn as you go. This package was created as an exercise in communication just as much as algorithm development. Having said that, there are some phrases I may use that one may not recognize. Use the Google. - Customizability. The software should facilitation integration of custom features as to fully empower the user to learn as much as possible. - Generalization. By avoiding limitations in our explanations, we allow the reader to not learn about a niche computer science problem but instead apply the knowledge gained to as many of their endevors as possible. You'll notice this repository doesn't include the phrase "Traveling Salesmen Probem".

To learn about the problem in detail, its best to provide a list of phrases used in this package and their documentation.

2 Vocabulary

- The problem: We can’t define the problem without first defining its components, such as nodes, costs, and paths. Recall that one core concept of this package is the ”teach as you go” philosophy. We’ll revisit the defination of the problem soon. - node: a point of interest. Since a node is a point, in the mathimatical sense, it has some defining characteristics. A point on a map of the globe may have two defining characteristics, longitude or lattitude. Likewise a point on an X, Y plot is defined by its X-coordinate and its Y-coordinate. Because we desire generalization, we’re going to call each point p_n where n is a counting number that denotes a unique indentifier of that node. For example, if I have three nodes, I could unique assign them each a name like p_1, p_2, and p_3. - cost - optimal path - optimal cost - distance matrix/2D distance arary - combinatorial optimization

(Boring) Vocabulary - The package: this is the phrase we’ll use to refer to the combination of documentation, code, and reccomended external resources. - The reader/The user: You!

Even more interesting is how

Finding the most desirable path between some locations is a general, previlent problem. This codebase facilitates finding the most desireable path among a set of locations, so long as information about those locations is provided. In this quanitative exploration, we refer to these locations as ”Nodes”, each having some position in space, such as longitude and latitude. The ”most desirable path” between nodes is normally the path that visits all the nodes, but minimizes some cost funciton, such as time taken to traverse the path or distance traveled along the path. In other words, we want to go everywhere but while costing us the least. We can generalize this problem and use an algorithmic approach to find the most desirable path.

This repository allows one to apply algorithms designed to quickly obtain the most desirable path. Such algorithms are useless without a set of nodes to test them on, so the feature to create sets of nodes with various characteristics is included. This codebase also makes it very easy to create and test user-created algorithms. To learn more about the code, read ‘The Codebase‘ section.

3 A Quantitative Description of the Problem

Given a set of N static nodes, each having p¡sub¿1¡/sub¿ and p¡sub¿2¡/sub¿ where 0 ¡= p¡sub¿1¡/sub¿, p¡sub¿2¡/sub¿ ¡= 1, find the minimum cost path that touches all N nodes, where the cost is defined as the 2 dimensional cartesian distance between any two nodes.

The Codebase Calculating the the minimum cost path through a set of notes is a computational taxing problem. One must perform on the order of N! calculations to find the optimal path through a set of N nodes. This becomes impractical on a standard Macbook when N ¿ 10. Estimating the optimal solution using intellegent algorithms is much more effective technique that normally results within 10

This software package allows users to rapidly develop and test algorithms to estimate the optimal path. With the code in this repository, one leverage the easy-to-use CLI to focus on the development of the interesting bits: the pathfinding algorithm itself. Users of this codebase can: 1. Create sets of nodes using ‘src/createmap.py‘. The placement of the nodes is based on some sort of 2D distriubtion. A user can create their own distribution function or use any of the built in ones available at ‘src/mapcreation/*.py‘. A distribution function can take various other parameters, for example ‘fixednumberofgroups‘ requires the user to specify how many groups to create. For examples of built in distribution functions, see ‘data/samplemaps/*.png‘. Currently the build in distribution functions are: 1. ‘randomuniform‘: randomly distribute nodes 1. ‘ball‘: a normal distribution in p¡sub¿1¡/sub¿ and p¡sub¿2¡/sub¿ centered at (0.5, 0.5) 1. ‘donut‘: centers the peak of a normal distriubtion some distance away from (0.5, 0.5) 1. ‘fixednumberofgroups‘: creates a handful of clusters randomly around the map 1. ‘sinusoidal‘: the distriubtion function is sin¡sup¿2¡/sup¿(p¡sub¿1¡/sub¿, p¡sub¿2¡/sub¿) 1. Calculate a path through the set of nodes using ‘src/execute.py‘. The path is calculated via one of the solvers available at ‘src/solvers/*.py‘. Again, a user can integrate their own solver. They’d do this by writting the code and putting it in the ‘src/solvers‘ folder and enabling it in the ”USAGE” section of the document string at the top of ‘src/execute.py‘. The included solvers are: 1. ‘brute‘: calculate the cost of all possible paths through a set and return the minimum cost path. This is the only surefire way to get the optimal path but is incredably slow when N is large. 1. ‘nearestneighbor‘: algorithm that, when at any given node, travels to the next closest node 1. ‘randomneighbor‘: when at any given node, randomly selects another unvisited node to travel to next. Expect this to be far form the optimal pathh through the city list. Do not return home to origin city after visiting every city.

Notes Some important notes: 1. Solutions visit all nodes 1. The first node in the cityLocations file is considered the origin. This is unchangeable. 1. From any node one can visit any other node as long as they assume the cost in the cost matrix 1. This is important because in some Traveling Salesmen Problems, not every node can visit each other node. We can account for this case by setting the cost of this path and its inverse (A-¿B has inverse B-¿A) to infinity in the cost matrix. By doing this we introduce the subcase where a set of nodes may be intrinsicly unable to travel to another set of nodes, resulting in the cost of the lowest cost path equal to infinity. 1. Once a path from A-¿B is taken, it and its inverse is removed from possible future paths to be taken. AKA no repeats. 1. To explain, consider set A, B, C, D. The path A-¿B-¿C-¿D is obvious, but the above statement disallows A-¿B-¿C-¿B-¿A-¿D. If we considered paths like this I believe there would huge, but finitely many paths to consider on a set of finite size.

Want To Contribute? The goal of this project is to create an infrastructure for estimating solutions of the problem. The infrastructure should: - allow for a user to easily create an randomly generated node map: - using premade algorithums - by creating their own map creation algorithm - allow for a user to easily apply a solution estimation algorithm to a node map: - using premade algorithms - by creating their own solution algorithm - visualize solutions to previously executed solution algorithms - easily apply various solution algorithms to maps created from various map creation algorithms

To Do: 1. Make a ”–demo” flag that a user can run immediatly upon cloning repo in order to get an idea for what this codebase can do 1. Automated test cases so when building a feature we can tell what fails and what passes. 1. Clear up why JSON is saved the way it is. Fix save method such that non serilizable objects (2+ dimenionsal arrays) play nice with JSON format requirements. 1. Update: curretly using ‘toList()‘ to make 2D arrays serializable. 1. Add functionality to define an origin node and to define the ability to have to end at that origin node. 1. Redo CLI. 1. Use YAML... 1. Can stoichastic branches help? 1. What about a ML algorithm? 1. Be able to easily create statistics using a ‘wrapper.py‘ like program about how different algorithms work on different types of maps. 1. What about cases where the distance Matrix changes over time? 1. Decide if distance matrix can be used to fully power algs working on a set of nodes. Do the algs need the actual city locations too?

Literature Cited