Some Title

Sean Hunter Brooks

seanhunterbrooks@gmail.com

Git: @astronomerhunter

September 29, 2017

Abstract

This paper and accompanying code allows the reader to perform quantitative analysis of a basic graph theory problem, create custom solutions complete with visualizations, and easily apply techniques and concepts to other computer science domains. The information is presented in such a way that little prior mathematical and coding knowledge is required.

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 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. But 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? This problem is historically described as the Traveling Salesmen Problem, or TSP, and is has been a computer science challenge since the 1930's.

The generalization of the TSP is prevalent in the world today. Package delivery, circuit board manufacturing, 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 repercussions of getting there. The software in this Github repository offers the investigative reader the ability to learn about various predictive techniques.

This Github repository:

1. walks the reader through the details of the problem at hand
2. provides sample algorithms that can be used to solve the problem
3. allows the user to create custom algorithms to solve the problem
4. creations visualizations of the problem and solution

Ultimately, this software and accompanying documentation was created with the following concepts in mind:

1. **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.
2. **Education**. Complex concepts are assemblies of simpler ones; learn as you go. This package was created as an exercise in communication just as much as algorithm de-

velopment. Having said that, there are some phrases I may use that one may not recognize. Use the Google.

3. **Customizability**. The software should facilitation integration of custom features as to fully empower the user to learn as much as possible.
4. **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 Simple Vocabulary

1. **The package**: The combination of resources referenced in this paper. A component of the package is the repository located here.
2. **The reader/The user**: You!
3. **Vertex**: A point of interest. Since a vertex is a point, in the mathematical sense, it has some defining parameters. A point on a map of the globe is normally said to have two defining parameters, longitude or latitude. Likewise a point inside a ballon can be uniquely defined by its X-coordinate, Y-coordinate, and its Z-coordinate. Because we desire generalization, we're going to call each point p_n where n is a counting number that denotes a unique identifier of that vertex. For example, if I have a **Set of Vertices** containing three vertices, I could unique assign them each a name like p_1, p_2, and p_3. Each point p_n may have m parameters which are described by p_n^1, p_n^2, ..., p_n^m.
4. **Path**: An ordered list of vertices. Should you be solving the TSP with three vertices, A, B, and C, the paths [A, C, B] and [B, C, A] are valid. The path [A, D, C, B] is not valid on the previously mentioned set of vertices because it includes D.
5. **Optimal Path**: The most desired path among a set of vertices, according to the problem statement.
6. **Edge**: Connection between two vertices. Each edge has a start vertex and an end vertex.
7. **Cost**: The penalty for incorporating an edge in a path. We don't restrict the domain of any costs, so zero and negative costs may exists.
8. **Cost Function**: A function C that operates on inputs to provide a cost value. Should the cost function be the cartesian distance in two dimensions, it would take a origin vertex and a destination vertex and maybe look like $C(p_o, p_d) = \sqrt{(p_o^1 - p_d^1)^2 - (p_o^2 - p_d^2)^2}$.
9. **Optimal Cost**: The cost of the most desired path.

10. **Graph**: Combination of a set of vertices and corresponding edges.

With these in mind, we can define the problem statement.

3 The Problem

Given a set of N vertices, each having p_n^1, p_n^2 where $0 \leq p_n^1$, $p_n^2 \leq 1$, find the path with the lowest possible cost that contains each vertex at least once. Calculate the cost between any two vertices as

$$C(p_1, p_2) = \sqrt{(p_1^1 - p_2^1)^2 - (p_1^2 - p_2^2)^2} \tag{1}$$

.

4 Computational Time

The only surefire way to solve the above problem statement is to determine that a path is the optimal path is to compare its cost against the cost of every other path. However, this is a computational taxing problem because there are on the order of $N!$ many unique paths for a graph with N vetices . It becomes impractical on a standard MacBook when $N > 10$.

Estimating the optimal solution using intelligent algorithms is much more effective technique that normally results within 10% of the optimal solution for the most pragmatic algorithms. To test the performance of those algorithms for a given graph, one can find the optimal cost via a brute force method and compare to the resulting cost of the path produced by the estimation algorithm.

5 The Code

The repository allows one to use included algorithms or to create new ones to determine or estimate the optimal path through a graph. Such algorithms are useless without a set of vertices to test them on, so a graph creation feature exists. In addition, a feature to create a .gif visualization of the resulting optimal path is included. All of the code is written in easy to read Python 2.7.

5.1 Graph Creation

There are several tools to create graphs included out of the box. They create static graphs with various vertex distributions. One can easily create their own graph making algorithm as well. The `src/create_map.py` module is responsible for creating `data/maps/MID.../` files, which are JSON representations of graphs. Each of those JSON files contains ...

The standard graph making algorithms are:

1. random uniform: randomly distribute vertices
2. ball: a normal distribution in p¡sub¿1¡/sub¿ and p¡sub¿2¡/sub¿ centered at (0.5, 0.5)
3. donut: centers the peak of a normal distribution some distance away from (0.5, 0.5)
4. fixed number of groups: creates a handful of clusters randomly around the map
5. sinusoidal: the distribution function is sin¡sup¿2¡/sup¿(p¡sub¿1¡/sub¿, p¡sub¿2¡/sub¿)

5.2 Included Algorithms

Once you have a graph created, you can use `src/execute.py` to run algorithms on it. Again, this software was designed with customizability in mind so its trivial to integrate a new estimation algorithm. Out of the box, 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.
2. nearest neighbor: algorithm that, when at any given vertex, travels to the next closest vertex
3. random neighbor: when at any given vertex, randomly selects another unvisited vertex 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.

6 Final Notes

These are mostly for the author and reflect the current state of the code.

1. Solutions visit all vertices
2. The first vertex in the cityLocations file is considered the origin. This is unchangeable.
3. From any vertex one can visit any other vertex as long as they assume the cost in the cost matrix
   (a) This is important because in some Traveling Salesmen Problems, not every vertex can visit each other vertex. 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 vertices may be intrinsicly unable to travel to another set of vertices, resulting in the cost of the lowest cost path equal to infinity.

4. Once a path from A-¿B is taken, it and its inverse is removed from possible future paths to be taken. AKA no repeats.
    (a) 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.

7 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 vertex and to define the ability to have to end at that origin vertex. 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 vertices. Do the algs need the actual city locations too?

Literature Cited