Investigations in Graph Theory

Sean Hunter Brooks

seanhunterbrooks@gmail.com

October 2, 2017

Abstract

This paper serves as an introduction to Graph Theory optimization problems. I define several Traveling Salesmen-like problems, investigate properties of the resulting graphs, perform quantitative analysis of various algorithms used to obtain optimal paths, and apply computer science techniques. Supplementary to this paper is an easy-to-use `Python` software package with the ability to create custom graphs, apply custom optimization algorithms, and visualize results. A fundamental intent of our work is to present information 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. In essence, it says "Find the optimal path through a graph." 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, the brute force method where one calculates all the possible routes and their cost. However, because of the immense computational drawback associated with this technique, the brute force algorithm is not ideal and in some cases infeasible. Because of this, many techniques have been developed to estimate the optimal path. 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 algorithms.

Ultimately, the 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 development. 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. Develop your own algorithms!

4. **Generalization**. By avoiding limitations in our explanations, I allow the reader to not learn about a niche computer science problem but instead apply the knowledge gained to as many of their endeavors as possible. You'll notice this repository doesn't include the phrase "Traveling Salesmen Problem".

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

1.1 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 2 identifying parameters denoted by x_n and y_n. Vertices can also be called nodes.

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. We'll denote edges from A to B as $\overline{AB}$.

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 $C(p_o, p_d) = \sqrt{(x_o - x_d)^2 + (y_o - y_d)^2}$.

9. **Optimal Cost**: The cost of the most desired path.

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

11. **Optimization Algorithms**:

12. **Distance Matrix**:

2 The Problems

Here's an example of an common problem statement:

> Given a set of N vertices, each having x_n and y_n where $0 \leq x_n, y_n \leq 1$, find the path with the lowest possible cost that contains each vertex at least once and starts at N_0. Calculate the cost between any two vertices as
>
> $$C(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{1}$$
>
> .

This basically says "Find the shortest path among some nodes in Cartesian space." However, we can complicate the problem in several ways.

1. **Existence of an Origin City**: One can require that the optimal path start and/or end at a certain vertex. In the above problem statement, there exists an origin node, but the optimal path doesn't need to travel back to it after visiting all the other nodes.
2. **Allowance of Repeat Visits**: One can allow or disallow if the optimal path can return to a vertex after already being there. Imagine the case where the sum of costs $\overline{AB}$ and $\overline{BC}$ is less than $\overline{AC}$. Should the optimization algorithm choose to take the $\overline{AB} + \overline{BC}$ route but has already visited B, we'd need to allow repeat visits of nodes.
3. **Complete Intra-Vertex Travel Allowed**: Can one travel from any one vertex to any other? If not, we can write that edge costs as the sum the optimal costs of traveling to that vertex through intermediary vertices. However, this normally requires the allowance of repeat visits. To be more clear, consider the case where the only way to get to A is from B. Once you travel to A, you'd be stuck if you couldn't travel back to B. In this case, A would have to be your last node visited. What if another pair of nodes share the property that A and B share. The solution becomes impossible in the case where repeat visits are disallowed.
4. **Evolving Cost Function**: Imagine that the cost function is dependent on not only the set of nodes and edges, but also the current path through that graph. An interesting cost function might be where the cost of any edge is equal to the length of the path up to that point.

2.1 The Brute Force Solution

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 a huge number of paths to consider.

Even if repeat visits are disallowed, there are on the order of $N!$ many unique paths for a graph with N vertices. 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.

3 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.

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

3.1 Graph Creation

Our graphs contain a set of edges and vertices. The distribution of vertices is the sole information needed to construct the graph. We sample from a probability density function PDF to define the distribution of vertex locations. From their locations we can calculate the cartesian distance, giving us both necessary elements of the graph.

All graphs restrict the possible locations of vertices to $[0, 1]$ in each dimension, thus the PDF does as well. Of course, the integration of P should always yield 1.

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 included vertex distributions are:

1. **Random Uniform**: A random uniform distribution spanning [0, 1] for each parameter.

2. **Ball**: A normal distribution centered on (0.5, 0.5) with provided standard deviation σ.

3. **Donut**: The peak of a normal distribution is some away from (0.5, 0.5) and is radially symmetrical. Provide that distance d and the standard deviation of the normal distribution σ.

4. **Groups**: Group nodes into a provided number of groups N_g where the center of each group is a node whose location is selected on a random uniform distribution from 0 to 1. For each node belonging to a group, calculate its location by sampling from a normal distribution with standard deviation σ centered on the center of that group.

5. **Sinusoidal**: Given

Their PDFs are:

1. **Random Uniform**:
$$PDF(x, y) = 1 \tag{2}$$

2. **Ball**:
$$PDF(x, y|\mu_x = 0.5, \mu_y = 0.5, \sigma_x, \sigma_y) = \frac{1}{2\pi\sigma_1\sigma_2}\frac{1}{\sqrt{1-\rho^2}}\exp\left(\frac{-z}{2(1-\rho^2)}\right) \tag{3}$$
where
$$z(x, y) \equiv \frac{(x-\mu_x)^2}{\sigma_x^2} - \frac{2\rho(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y} + \frac{(y-\mu_y)^2}{\sigma_y^2} \tag{4}$$
and
$$\rho = \frac{V_{x,y}}{\sigma_x\sigma_y} \tag{5}$$
where $V_{x,y}$ is the covariance.

3. **Donut**:

4. **Groups**:

5. **Sinusoidal**:

3.2 Optimization 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 path through the city list. Do not return home to origin city after visiting every city.

4 Investigations

4.1 The Evolving Graph

The distance matrix changes as nodes are visited...

4.2 Average Cost of Random Path on a Random Uniform Graph with N Nodes

For random neighbor on random uniform: Number of Nodes, Number of Samples, Average Path Length, Variance of average path length 100, 5, 1.99989227691, 0.254139935131

How does random neighbor scale with N? Linearly?

4.3 Average Cost of Optimal Path on a Random Uniform Graph with N Nodes

Does this increase tapper off as N becomes huge?

5 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
2. Automated test cases so when building a feature we can tell what fails and what passes.
3. 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.
4. Update: curretly using 'toList()' to make 2D arrays serializable.
5. Add functionality to define an origin vertex and to define the ability to have to end at that origin vertex.
6. Redo CLI.
7. Use YAML...
8. Make algs to randomly explore short predictive branches to predict next step.

9. Make algs to deduce properties of graphs and apply a ML alg to it.
10. Be able to easily create statistics using a ‘wrapper.py‘ like program about how different algorithms work on different types of maps.
11. What about an evolving graph
12. 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?