Google Summer of Code

# NetworkX: Implementing the Asadpour Asymmetric Traveling Salesman Problem Algorithm

Matt Schwennesen
mjschwenne@gmail.com

13 April 2021

## 1 Abstract

The traveling salesman problem is one of the most famous problem in graph theory, first proposed about 90 years ago. It is roughly stated as "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" which is then travelled by a hypothetical salesman. The asymmetric traveling salesman problem (ATSP) is the variant in which the costs between the cities are different depending on which cities is the origin and destination. This problem is NP-complete, so we have to approximate the solution in order to complete the problem in anything approaching reasonable time. The best known algorithm to approximate a solution to the asymmetric traveling salesman problem was developed by Asadpour *et al.* [9] and produces an $O(\log n / \log \log n)$ approximation of the optimum solution. This project seeks to implement the Asadpour algorithm within NetworkX, utilizing existing parts of the project and its dependencies if they are available.

## 2 Technical Details

The Asadpour algorithm has similarities to Christofides' algorithm for the symmetric traveling salesman problem, which is already implemented in NetworkX pull request #4607, as a $\frac{3}{2}$ approximation of the minimum cost Hamiltonian cycle. Christofides' algorithm uses a minimum spanning tree of a simple graph as the basis for the Hamiltonian cycle, but as I briefly discussed with Dan Schult in discussion #4705, there is no good definition of a minimum spanning tree for a directed graph like what is used in the Asadpour algorithm.

Instead of trying to define one, the Asadpour algorithm creates a different kind of spanning tree called a $\lambda$-random spanning tree with additional special properties such as thinness in order to be used in a similar manner. As was also discussed in #4705, this is an area

in which functionality beyond the scope of implementing this algorithm could be added to NetworkX, as the ability to generate $\lambda$-random trees has other uses.

Code in pull request #4607 outlines the usage requirements for my implementation of the Asadpour algorithm. In `algorithms.approximation.traveling_salesman.py`, the overarching method is `traveling_salesman_problem()`, which takes in a method corresponding which traveling salesman problem algorithms use. Thus, the new Asadpour method must be compatible with `traveling_salesman_problem()`. Fortunately, all of the other methods in `traveling_salesman.py` also use a complete graph of the shortest paths between each pair of vertices, found with `nx.all_pairs_dijkstra()`. This is exactly how the Asadpour algorithm operates and it enforces the triangle inequality, another requirement of the algorithm. The Asadpour algorithm is well suited to be integrated into the existing infrastructure NetworkX has in `traveling_salesman.py`.

The algorithm starts by solving the Held-Karp relaxation of the ATSP, defined by the following linear program in section 3 of both Asadpour papers [9, 1].

$$\begin{aligned}
\min \quad & \sum_{a} c(a)x_a & & (1)\\
\text{s.t.} \quad & x(\delta^+(U)) \geqslant 1 & \forall\, U \subset V \text{ and } U \neq \varnothing & (2)\\
& x(\delta^+(v)) = x(\delta^-(v)) = 1 & \forall\, v \in V & (3)\\
& x_a \geqslant 0 & \forall\, a & (4)
\end{aligned}$$

The recommended way to solve this linear program, as per the paper, is with the ellipsoid algorithm, the first polynomial timed linear programming algorithm. My first way to solve the Held-Karp relaxation was to rewrite the linear program into standard form and use `scipy.optimize.linprog()` to actually find the solution. The first blog post that I wrote was about how I would be able to convert the relaxation into standard form. Upon trying to approximate the size of the coefficient matrix $A$, I realized that this program is massive, too large to write in standard form. For a complete bi-directed graph, there would be about $1.606 \times 10^{60}$ entries in $A$. A new solution to the Held-Karp relaxation was needed, and this is where the ellipsoid algorithm really steps in. Rather than being a choice about time complexity it was a choice about space complexity. None of NetworkX's dependencies provide at way to perform the ellipsoid algorithm, so it will be necessary to write one for this project.

In the next phase of the algorithm, an exponential probability distribution on the spanning trees of $G$ is approximated using weights $\{\tilde{\gamma}\}_{e \in E}$ to within $\epsilon = 0.2$. These $\tilde{\gamma}$ can be found with the algorithms in sections 7 and 8 of [1]. The algorithms discussed are interchangeable, but in the interest of providing a working solution first, I would implement the one discussed in section 7, with the option to improve this part of the algorithm by replacing it with the ellipsoid version in section 8 if there is enough time. The combinatorial algorithm is roughly

1. Set $\gamma = \vec{0}$
2. While there exists an edge $e$ with $q_e(\gamma) > (1 + \epsilon)z_e$:

- Compute $\delta$ such that if we define $\gamma'$ as $\gamma'_e = \gamma_e - \delta$, and $\gamma'_f = \gamma_f$ for all $f \in E \setminus \{e\}$, then $q_e(\gamma') = (1+\epsilon/2)z_e$.
- Set $\gamma \leftarrow \gamma'$.

3. Output $\tilde{\gamma} := \gamma$.

As listed on page 16 of [1]. $\delta$ can be computed using the formula below, derived from lemma 7.1 using the fact that $q_e(\gamma') = (1+\epsilon/2)z_e$.

$$\frac{1}{(1+\epsilon/2)z_e} - 1 = e^{\delta}\left(\frac{1}{q_e(\gamma)} - 1\right) \tag{5}$$

$$\ln\left(\frac{1}{(1+\epsilon/2)z_e} - 1\right) = \ln\left(e^{\delta}\left(\frac{1}{q_e(\gamma)} - 1\right)\right) \tag{6}$$

$$\ln\left(\frac{1}{(1+\epsilon/2)z_e} - 1\right) = \ln\left(e^{\delta}\right) + \ln\left(\frac{1}{q_e(\gamma)} - 1\right) \tag{7}$$

$$\ln\left(\frac{1}{(1+\epsilon/2)z_e} - 1\right) = \delta + \ln\left(\frac{1}{q_e(\gamma)} - 1\right) \tag{8}$$

$$\ln\left(\frac{1}{(1+\epsilon/2)z_e} - 1\right) - \ln\left(\frac{1}{q_e(\gamma)} - 1\right) = \delta \tag{9}$$

$$\ln\left(\frac{\frac{1}{(1+\epsilon/2)z_e} - 1}{\frac{1}{q_e(\gamma)} - 1}\right) = \delta \tag{10}$$

$$\ln\left(\frac{\frac{1-(1+\epsilon/2)z_e}{(1+\epsilon/2)z_e}}{\frac{1-q_e(\gamma)}{q_e(\gamma)}}\right) = \delta \tag{11}$$

$$\ln\left(\frac{1-(1+\epsilon/2)z_e}{(1+\epsilon/2)z_e} \times \frac{q_e(\gamma)}{1-q_e(\gamma)}\right) = \delta \tag{12}$$

$$\ln\left(\frac{(1-(1+\epsilon/2)z_e) \times q_e(\gamma)}{(1+\epsilon/2)z_e \times (1-q_e(\gamma))}\right) = \delta \tag{13}$$

While unsightly it is certainly feasible. Using `numpy.array` or python dictionary to store the vector $\gamma$ and Kirchhoff's matrix tree theorem with a weighted Laplacian, the values of $q_e(\gamma)$ can be calculated as in section 5.3 of [1]. Since the values of $\lambda_e$ needed to create the Laplacian is $e^{\gamma_e}$, this matrix needs to be recalculated for each iteration of the algorithm in section 7. As such python will certainly be able to support this iterative approach to finding $\{\tilde{\gamma}\}_{e\in E}$.

The algorithm then samples $2\lceil \ln n \rceil$ spanning trees from $\tilde{p}(.)$, the distribution approximated using $\tilde{\gamma}$. Here, $\tilde{p}(T) = \frac{1}{P}\exp(\sum_{e\in T}\tilde{\gamma}_e)$ for all $T \in \mathcal{T}$ where $P = \sum_{T\in\mathcal{T}}\exp(\sum_{e\in T}\tilde{\gamma}_e)$, as given in theorem 5.2 of [1]. After $\tilde{\gamma}$ is found, $P$ would only need to be calculated once. Each tree is sampled using Kirchhoff matrix tree theorem and an iterative algorithm where the probability of adding an edge is conditioned on the other added edges. This once more uses the weighted Laplacian described in section 5.3 of [1] and a sampling method similar to *Generating Random Combinatorial Objects* [3]. *Generating Random Combinatorial Objects* [3] is not available online, but the Michigan Technological University library had a copy which I was able to scan and read. The premise is to repeatedly contract edges of $G \in T$ to

generate a new Laplacian and thus find the new probability that the edge is in $T$ as outlined in ALGORITHM A8 of [3]. Using a `numpy.array` as the matrix, there are plenty of support functions in `numpy` to facilitate the required operations such as deleting a row and column and taking the determinate of the resulting matrix to find a cofactor to use Krichhoff's matrix tree theorem with. Unfortunately ALGORITHM A8 this a time complexity of $O(|V|^3|E|)$ [3], however this is still better than calculating the probability of each possible spanning tree as there are in exponential number of such trees.

For each of the sampled trees, which are undirected. If the undirected trees are placed in a list which is heapified using the build in `heapq.heapify()` [7] function in the python library the smallest cost tree can be found in linear time. The minimum cost undirected spanning tree is then directed so to minimize the weight of each arc. From here, use Hoffman's circulation theorem to find the minimum flow in the complete bi-directed graph which contains that directed tree, $\vec{T}^*$. This can be accomplished using an existing NetworkX function, `algorithms.flow.network_simplex()` [5]. Find the Eulerian circuit using `algorithms.eulalrian_circuit` already in NetworkX [4] and shortcut it using `_shortcutting()` in `algorithms.approximation.traveling_salesman.py` of pull request #4607. The output of `_shortcutting()` will be the $O(\log n / \log \log n)$ approximation of the asymmetric traveling salesman problem.

## 2.1 Summary of New Methods

Several new method would need to be implement within NetworkX, a tentative outline is given.

- In `algorithms.approximation.traveling_salesman.py` or possibly the creation of `algorithms.tree.lambda_random.py` to contain some of them.
  - `asadpour(G, weight=''weight'')` → This will be the overarching method of the Asadpour algorithm within NetworkX, similar to `christofides()` already in pull request #4607. This method will manage and interact with all of the other methods listed here to produce the approximate solution to the ATSP.
  - `held_karp(G, weight=''weight'')` → Converts the general Held-Karp relaxation into standard form for the specific complete bi-directed graph $G$. The relaxation is then solved using `scipy.optimize.linprog()` [8] and returns the optimum value of the objective functions as will as the fractional values of the arcs in that optimum solution.
  - `spanning_tree_distribution(z)` → Uses $z$, the scaled vector from the Held-Karp relaxation to find $\{\tilde{\gamma}\}_{e \in E}$. This method will require `laplacian()` to help generate $q_e(\gamma)$.
  - `sample_spanning_tree(G, gamma)` → Returns a $\lambda$-random spanning tree using ALGORITHM A8 in [3] with $\lambda_e = e^{\gamma_e}\ \forall\ e \in E$. This method will also need `laplacian()` to help generate a $\lambda$-random spanning tree.

– `laplacian(G, gamma=‘‘gamma’’)` → Returns the value of a cofactor for the weighted Laplacian matrix defined as

$$L_{i,j} = \begin{cases} -\lambda_e & e = (i,j) \in E \\ \sum_{e \in \delta(\{i\})} \lambda_e & i = j \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

This is an opportunity to use an existing method within NetworkX, namely `networkx.linalg.laplacianmatrix.laplacian_matrix()`. We generate the matrix defined in (14) using this method, which returns a SciPy sparse matrix that we immediately convert into a standard `numpy.array` [6] using `toarray()`. Then this laplacian methods is a wrapper to find the first cofactor of the returned matrix as it is used in multiple places in the Asadpour algorithm utilizing standard methods in scipy and numpy to find the cofactor.

## 2.2 Testing

Testing is a critical part of developing solid software and this project should be no exception. Each of the proposed methods should be rigorously tested before the project is integrated into NetworkX as a whole. That testing will have the following form.

For `asadpour` it is impractical to brute form a solution for the ATSP to check that the approximation is within. However, we know that the lower bound of the cycle returned is the solution of the Held-Karp relaxation. Thus testing this method would consist of two steps, making sure that the returned cycle is a Hamiltonian cycle through the graph (that is, it is a feasible solution) and that the total cost of the cycle is within $\log n / \log \log n$ of the Held-Karp solution, $\text{OPT}_{\text{HK}}$.

`held_karp` will use an ellipsoid solver that I will have to write, but for small samples it can be checked with `scipy.optimize.linprog()` [8]. Using the approximation of the coefficient matrix from my blog post, a graph with 10 vertices will have about 1,163,016 entries which is at least feasible. Reducing this to nine vertices creates a matrix with about 309,520 entries. While these are certainly larger than we would want in regular operation of the algorithm, it will provide an oracle to check the results of `held_karp` against.

The critical property of the distribution created by `spanning_tree_distribution` is that $\sum_{T \in \mathcal{T}: T \ni e} \tilde{p}(T) \leqslant (1+\epsilon) z^*_e$. This will rely on the fact that `held_karp` is working appropriately, but that properly can be verified with a series of loops for a few distributions.

Testing `sample_spanning_tree` will build off of `spanning_tree_distribution`. If the test samples numerous spanning trees for a given distribution, a goodness of fit test will evaluate how closely the sampled trees match the input distribution.

Finally for `laplacian`, this method also relies on other methods within NetworkX and its dependencies to compute the first cofactor of the laplacian matrix. It can be verified by using another piece of software such as MATLAB once the matrix is generated (most likely by hand) to process the same data from a graph.

# 3 Schedule of Deliverables

## 3.1 Application Deadline & Community Bonding (13 April - 7 June)

During this time, I will ensure that my development environment is properly configured and continuing my research on the topics involved in this project. I will continue to research areas of the algorithm that I am not confidant in, particularly how to implement the ellipsoid algorithm, using resources such as the Michigan Technological University library and my graph algorithms or combinatorics professors. The application template states that I will have to write blog posts during this time, so I have already configured a blog site using Jekyll and GitHub pages which is accessible at mjschwenne.github.io.

During this time I will work on a smaller project for NetworkX to ensure that I am familiar with the project. In particular, I noticed two open issues on the GitHub page which are of interest, #4089 and #3107. Both impact `laplacianmatrix.py`, which I plan on using in this project, so I have an interest in making sure these methods are correctly implemented and contiguous with the rest of the project.

Finally, I will also create stub methods and draft the docstring for them. Data structures in each method will be carefully considered, in particular how data flows from one method to the next step in the algorithm. For large graphs it will be important to not have to recalculate information and extraneous information should not be passed between methods.

## 3.2 Phase 1 (7 June - 16 July)

The focus of phase 1 will be setup for the algorithm and the Held-Karp relaxation. The Held-Karp relaxation, defined in section 3, will be solved using the ellipsoid algorithm.

The first method to be implemented will be `held_karp`, currently planned for 7 June - 30 June. This three and a half week time frame should allow for the completion of the ellipsoid algorithm as well as the development of test cases. A full two and a half weeks will be spend implementing the method and the last week testing as many cases as I can. It is very important that this method works well as the rest of the algorithm cannot operate without it.

The next week, 30 June - 4 July, will be spent developing and testing `laplacian`. This method has to be completed before work can begin on either `spanning_tree_distribution` or `sample_spanning_tree`. `laplacian` is arguably the most straight forward of the proposed methods as it uses the most existing code, but it is vital that it is implemented correctly. The four days of allocated time should be enough to complete development and testing.

Implementation of the `spanning_tree_distribution` will occur over 5 July - 18 July. The first 7 days (5 - 12) will be spent developing the algorithm itself, as outlined on page 16 of [1] and the next week on testing the algorithm. Some time in this period will be spend on the phase 1 evaluation which is due starting 12 July. This will take the project to the end of phase 1.

A condensed list of dates is provided below.

- 7 June 2021 — Beginning of phase 1.
- 30 June 2021 — Held-Karp relaxation can be solved for the given graph $G$.
- 4 July 2021 — Find the cofactor of the weighted Laplacian matrix which is equal to $\sum_{T\in\mathcal{T}}\prod_{e\in T}\lambda_e$ for any graph $G$ using a correctly defined Laplacian.
- 12 July 2021 — Prepare phase 1 evaluation.
- 18 July 2021 — Calculate $\tilde{\gamma}$ to approximate the exponential distribution of spanning trees of $G$.

## 3.3 Phase 2 (16 July - 16 August)

Sample spanning trees using the approximate distribution, complete the Eulerian walk of $T^*$, then shortcut it to find the final approximation of the solution to the asymmetric traveling salesman problem, as well as continuing testing.

After completing `spanning_tree_distribution`, the next two weeks (19 July - 1 August) will be spent working on sampling spanning trees from distribution using the method `sample_spanning_tree`. This method will be templated off of Algorithm A8 from [3], but looking at that algorithm I believe that saving the value of $a'$ if $Z \leqslant a_j a'/a$ in order to reuse it in the next iteration will reduce the recalculations performed. The last 5 days of this window will be spent developing tests for this method.

The last two weeks of phase 2, 2 August - 15 August, will be spent ensuring that all of the component methods work to together in `asadpour`. Here all of the earlier testing will help reduce the amount of debugging needed to complete the Asadpour algorithm and facilitate the final testing. I expect that the final implement of the Asadpour method will be complete by 6 August which will allow the final 9 days of phase 2 to be devoted to final testing of the method.

A condensed list of date is below

- 1 August — Sample spanning trees of $G$ in accordance with the approximated probability distribution.
- 15 August — Be able to find $O(\log n/\log\log n)$ approximations for the asymmetric traveling salesman problem.

## 3.4 Final Week (16 August - 23 August)

My current time table has all development and coding in the project completed by the start of the final week. However, there are certainly unforeseen challenges which lay before me and having an extra buffer week will be helpful to ensuring that the project is completed by the end of the Summer of Code. Work may need to continue into this week, I am not planning on needing this time.

The real activity going on in this week should be preparing the for the final evaluation, due at the end of the week, and preparing the pull request to be merged into main. The docstings for the proposed methods will be updated to include examples, references and notes to be ready for sphinx to generate the documentation for the next release of NetworkX.

# 4 Development Experience

This will be my first contributions to an open source project, but I have used NetworkX in two projects.

I am currently taking MA 4208 Graph Algorithms / Optimization at Michigan Technological University, which teaches fundamental graph algorithms using [2]. While there is no programming assignments in the class, I decided to program the algorithms taught in the class in python using NetworkX.

A few of the algorithms are listed below, and all of my source code is in my GitHub repository.

- Dijkstra's Algorithm
- Hungarian Algorithm
- Ford-Fulkerson Algorithm

This project allowed me to get familiar with NetworkX and python as a whole. The last three weeks of this course focuses on linear programming and the simplex method (chapter 17 in [2]), so an implementation of the simplex method will be added before 1 May 2021.

My other project is actually part of another class that I am taking this semester, CS 3141 Team Software Project. This is a very unusual class where students are broken into small teams and they are then able to select a project of their own choosing to complete by the end of the semester. The course is graded not on if you complete the project but rather how well the team works together and it uses two week development sprints with weekly reports to evaluate us. My team's project, Atlas, is a fantasy city map generator which we are developing using python as well.

I have spent much of my time on Atlas working with NetworkX graphs as well. We generate a Voronoi diagram which is then translated into a NetworkX graph where each node is a point where three or more of the Voronoi regions. To facilitate this, I implemented the algorithm for chordless paths algorithm discussed by Takeaki Uno and Hiroko Satoh in An Efficient Algorithm for Enumerating Chordless Cycles and Chordless Paths [10]. While this algorithm is no longer a part of the Atlas project, my implementation can be found in my graph algorithms project discussed above.

This project's GitHub repository is here.

# 5 Why this project?

There are several reasons why this project is of interest to me. Primarily, I find the traveling salesman problem as a whole to be quiet interesting and more importantly practical. This is a problem where understanding the theory can directly be used to solve a practical problem.

Additionally, the Asadpour algorithm itself presents several features which I do not normally encounter in coursework but will be important for me later. It uses multiple area of mathematics, from graph theory to linear programming to statistics this algorithm draws on knowledge from many areas and I do not normally encounter in coursework. While each of these disciplines within mathematics is taught separately, that is not the case outside of the classroom and I have always enjoyed trying to connect topics in classes together. The original research paper does not contain an implementation of the algorithm, it actual or pseudocode. In my other coursework, when asked to implement an algorithm we are almost always given pseudocode of it, and I am looking forward to the challenge of having to develop an implementation for this algorithm from the original paper without any. This is how new algorithms are designed and then move to being used on concrete datasets and graphs and it is a process that I am very interested it.

# 6 Personal Information


- Name: Matt Schwennesen
- Email: mjschwenne@gmail.com or mjschwen@mtu.edu
- Phone: 734-972-1200
- GitHub: mjschwenne


I am an undergraduate student at Michigan Technological University majoring in Computer Science with a minor in Mathematical Sciences. Within mathematical sciences, I am particular interested in discrete mathematics and graph theory. I am also interested in data science and human interactions with data. While working at the computer science learning center on campus, I have come to realize there are many approaches to any problem and that you must always be willing to re-evaluate your approach to achieve the best results. I enjoy tackling difficult problems and believe that people are most creative when given problems that push their knowledge.

# References


[1] A. Asadpour, M. X. Goemans, A. Madry, S. O. Gharan, and A. Saberi, *An o(log n/log log n)-approximation algorithm for the asymmetric traveling salesman problem*, Operations research, 65 (2017), pp. 1043–1061, https://homes.cs.washington.edu/~shayan/atsp.pdf.

[2] W. Kocay and D. L. Kreher, *Graphs, Algorithms, and Optimization*, vol. 90 of Discrete mathematics and its applications, Chapman and Hall/CRC, Boca Raton, 2017.

[3] V. Kulkarni, *Generating random combinatorial objects*, Journal of algorithms, 11 (1990), pp. 185–207.

[4] NetworkX, *networkx.algorithms.euler.eulerian_circuit*. online, 2021, https://networkx.org/documentation/stable//reference/algorithms/generated/networkx.algorithms.euler.eulerian_circuit.html.

[5] NetworkX, *networkx.algorithms.flow.network_simplex*. online, 2021, https://networkx.org/documentation/stable//reference/algorithms/generated/networkx.algorithms.flow.network_simplex.html.

[6] NumPy, *numpy.array*. online, 2021, https://numpy.org/doc/stable/reference/generated/numpy.array.html.

[7] Python, *heapq - heap queue algorithm*. online, 2021, https://docs.python.org/3/library/heapq.html.

[8] SciPy, *scipy.optimize.linprog*. online, 2021, https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html.

[9] Society for Industrial and Applied Mathematics, *An O (log n / log log n )-approximation algorithm for the asymmetric traveling salesman problem*, SODA '10, Society for Industrial and Applied Mathematics, 2010, https://dl.acm.org/doi/abs/10.5555/1873601.1873633.

[10] T. Uno and H. Satoh, *An efficient algorithm for enumerating chordless cycles and chordless paths*, in Discovery Science, Lecture Notes in Computer Science, Cham, 2014, Springer International Publishing, pp. 313–324, https://doi.org/10.1007/978-3-319-11812-3_27.