8-20-2021

Finding Lonely Routes for Runners and Bikers

Ethan T. Spicher
Eastern Mennonite University

Finding Lonely Routes for Runners and Bikers

Ethan Spicher
ethan.spicher@emu.edu
August 20, 2021

*With help from Christof Teuscher, Philippe Proctor, The Portland State University REU Program and The National Science Foundation

Link to Code: https://github.com/EthanSpicher/glowing-happiness

---

Abstract

With COVID-19 raging around the world, personal health is even more important to a lot of people. One way to maintain good physical and mental health is to exercise according to Deslandes [2]. When exercising it may be important to make sure that you are running/biking on trails that are less populated than others, as well as taking into account the distance. This can be solved by creating an algorithm that allows the user to choose the starting and end point, and the algorithm will then find the optimal path between the two points with the distance and popularity of paths taken into consideration. This allows the user to get a nice workout in while minimizing possible exposure to others.

---

1 Introduction

With personal health becoming so important, it's critical that in a time such as COVID-19 there are ways that everyone can exercise without worrying about possible exposure. According to Andréa Deslandes [2], there is a strong relationship between physical activity and mental/physical health. By being active, it is beneficial to your mental/physical strength and that is so valuable in such a mentally and physically draining time of life. This

leads us to the goal of this research which is providing Portland and the surrounding area with a way to find better routes for biking and running to reduce the possible risk of transmitting COVID-19. By creating an algorithm that finds the best path from point A to point B, it would allow the user to go on a run/bike ride, as well as provide a sense or safety from contracting COVID-19.

This algorithm should be user friendly, accurate, efficient, and reliable. It would be ideal for this to be a standalone app, or an add on to other workout tracking applications like Strava or Garmin. The initial plan was that we would take the information form Strava.com heat maps (Figure 1), and then combine that information with data on the roads and trails. This would allow us to have more accurate data leading to more usable results. Although we ended up not using data from Strava's heat maps, that is still a goal in future work.

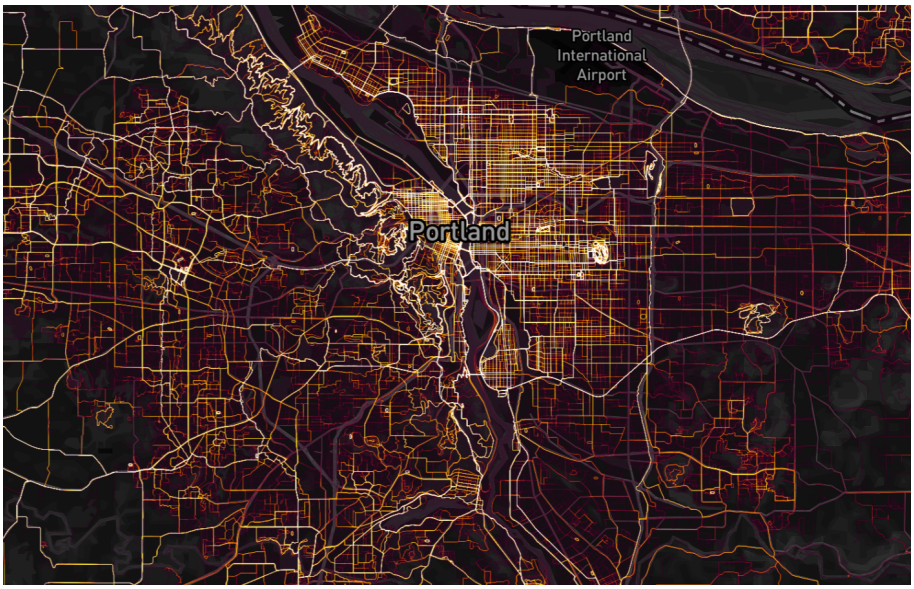

Figure 1: Strava Heat Map (Portland, OR)

The algorithm used in this research is the A*Prune algorithm that was created by Gang Liu and Ramakrishnan [3]. This algorithm allows the user to put in the starting point and ending point, and then it will find the best route while taking into account the distance and popularity of each road or path.

The following sections go over other work that is similar as well as work that has been fundamental in our research. Sections also go over the approach that we had, the results from the algorithm, and an overall summary of the whole research project.

2 Related Work and Background

Although this exact research hasn't been done before, there have been many other research topics that are similar. One of these topics is Elling Payne's work on reducing personal pollution exposure [4]. His research uses the A*Prune algorithm [3] to find the optimal path from point A to B while reducing pollution exposure and limiting the length of the path. This allows the user to find a path that might be a little bit longer then the shortest path, but that's the sacrifice that you have to be willing to take to reduce your exposure to pollution.

Another paper that was fundamental in allowing us to conduct this research was the A*Prune Algorithm paper by Gang Liu and Ramakrishnan [3]. This paper provides extensive information on the A*Prune algorithm. The A*Prune algorithm is a combination of the A* search algorithm and a pruning method, with the goal of finding the most efficient (shortest) between two points. This is done by finding the shortest path, while also pruning away the edges (paths) that the algorithm deems not useful. The way that the algorithm prunes the graph is that it expands the graph into all of the potential paths and cuts away the paths that don't connect the points and/or do not satisfy the constraints. This article then goes into comparing the A*Prune algorithm with other algorithms that have similar uses. The consensus was the A* algorithm is comparable to the other algorithms when it comes to running time, but is much more robust in the sense that it is able to run with more constraints and can consistently perform better.

Some other articles that where also looked at during this research includes Carlyle, et al paper on Lagrangian relaxation with the goal of finding the shortest constrained path [1]. This article shows a new algorithm that can solve the constrained shortest path problem. The constrained shortest path problem or CSPP involves finding the optimal path from one point to another while there are multiple different weights/constraints on each edge. The algorithm is applied to many different network models and efficiency is compared to other algorithms that perform similar tasks. This algorithm

is one of a kind, in that it uses Lagrangian mathematics to optimize the potential shortest path. One benefit of this algorithm over others is that it will eliminate edges that are proven to not lie on the optimal path. This is done by evaluating the graph before optimization, and removing unnecessary edges and nodes. By doing this, it greatly reduces the execution time. In conclusion, the authors have shown their proposed algorithm is effective in solving the constrained shortest path problem.

Now getting into how we decided to use the algorithm we did, we first looked at other algorithms that eventually lead us to the A*Prune algorithm. The first algorithm we looked into was Dijkstra's algorithm. Dijkstra's algorithm is an algorithm that allows the user to implement a weighted network graph $G = (v, u)$ where v represents the nodes (intersections), and u represents the edges (roads/paths). The algorithm then finds the shortest path from the starting point to all the other points. Now this algorithm is great when you only have one weight on each edge, but if you are trying to find the optimal path within a network graph with more then one weight per edge this is not the algorithm for you.

In our case we are wanting to be able to determine the optimal path to take when there is more than one weight, which in this case is the distance and the popularity of the path/road. By having these as our weights, we can find the path that has the shortest path, as well as the least number of other people. The algorithm that we found that allows us to have more than one set of weights is the A*Prune algorithm by Gang Liu and Ramakrishnan [3].

This algorithm is a combination of the A*Search algorithm, and a pruning algorithm, that when combined allows the user to find the optimal path easily and efficiently. We are also able to change the overall influence that the weights have on the final result. For example if you are going on a run, and are more interested in taking the shortest path, and not really worried about if other people are also using those trails, then the algorithm can be changed to find a path that fits your desire the best.

The way that the A*Prune algorithm works is very similar to how the A*Search algorithm works with a small difference. The A*Search algorithm works like this, it starts at the beginning node and looks around at its neighbors, and records the A*Search value for each neighbor. The A*Search value is the summation of the total distance traveled to get to that point plus the heuristic value at that point. The heuristic value is the distance from a point to the final point if measured by a bird. This means that the heuristic is always the same or smaller then the actual distance from that point to the

destination point. Once all of the neighbors A*Search values are calculated, the next node becomes the node with the lowest heuristic value. This is then repeated over and over again until the final node is reached.

This is better then Dijkstra's algorithm because it is faster, more efficient, and more applicable to our problem. Now that A*Search has been explained, we can go into detail on how the A*Prune algorithm works. Its really quite similar to the A*search algorithm, the difference is that when the A*Prune algorithm is executing each iteration, the algorithm also prunes away at the edges and nodes that are deemed to be not needed. This greatly reduces the execution time and improves efficiency drastically. Another wonderful part of the A*Prune algorithm is that it allows the user to input network graphs that contain multiple constraints for each edge, and that is precisely why we chose this algorithm for this problem.

3 Methodology

As stated in the introduction, the algorithm used in this article is the A*Prune algorithm created by Gang Liu and Ramakrishnan [3]. This algorithm was the heart of the whole algorithm driving our research. This algorithm did the computing on what path was the best to take, with the edge weights. Once we had decided on using the A*Prune algorithm, I then began to build the code around this algorithm so that it worked with the data that we wanted to provide it. Since our goal was to find the best path with the distance and popularity in mind, we had two different values paired with each edge. One value represented the distance, while the other value represented the busyness of that path (Figure 2).

The first step was to create an equation that combined the values of the distance and the weight together so that the algorithm could use that data. This was done by using this equations that Payne [4] also used. $w = s*d+(1-s)*p$ where d (distance) represented the length of each edge, p (popularity) represented the busyness of each edge, and s (strength) represents a value between 0 and 1. In this research strength was always 0.5. This meant that the distance and popularity had the same "weight" on the final result.

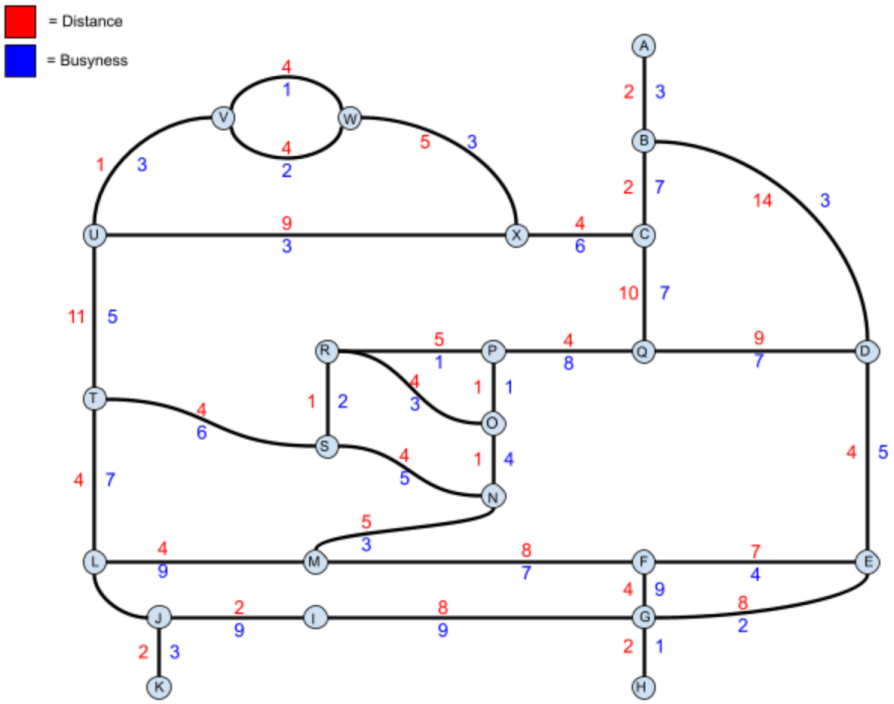

Figure 2: Network Graph Example

Once we had the weight for the edges figured out, the next goal was to provide data for the algorithm to make sure that it is working as expected. The data that I used for the initial tests was the data from Figure 2. This data proved that the algorithm was working properly. Since the algorithm was working as expected, the next step was to provide a way for the graph to be seen. This was solved by using NetworkX, a package in Python that allows you to do a lot of different things with network graphs and networks in general, but most importantly it allowed us to visualize the network. Although this was a step in the right direction, the visuals that NetworkX would generate where really quite hard to follow and not very useful especially with really large networks.

Now the way that we where able to obtain data from roads and trails that exist was to used OSMnx, witch is a package in Python that accesses data from Open Street Maps. This means that any streets or trails that have been added to Open Street Maps can be opened in Python. This package solved multiple problems for us. The first one is that it allows us to use real data and second it allowed us to generate much more understandable renders like Figure 3.

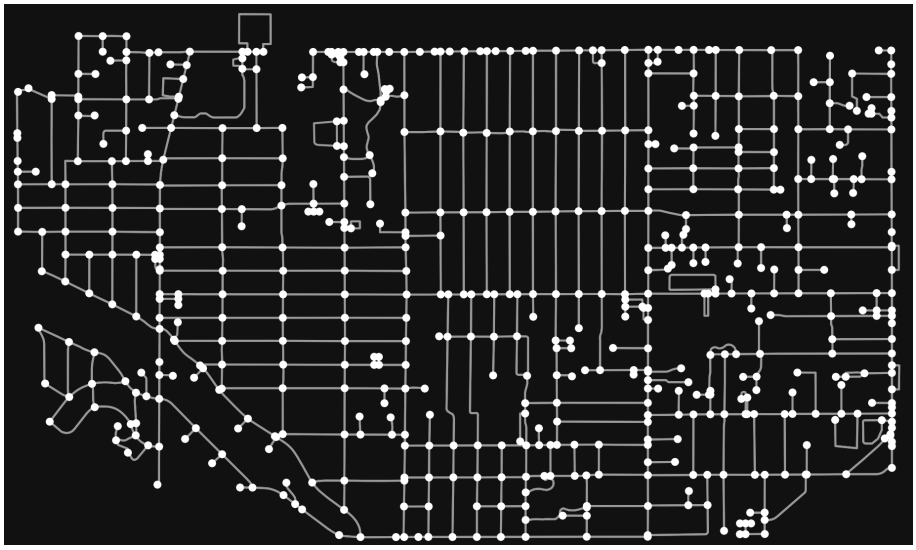

Figure 3: Brentwood - Darlington, Portland, Oregon, USA

This render shows the neighborhood of Brentwood - Darlington located in Portland Oregon. This is the network graph that we then used for the rest of the research project.

Once I figured out how to generate renders of this neighborhood, I began to work on extracting the data from OSMnx, like the length of the edges as well as the node IDs. After I figured out how to access that information, I began to convert this data so that the algorithm could work with it. Now that the data is usable I began to work on creating synthetic data to represent the popularity of each node. Initially I randomly assigned values for each edge, test the algorithm with different starting and end node combinations. After realizing that this wasn't ideal for testing, I began to try and generate data that made the edges closer to the middle of the render more popular. I was able to get this to work, and the renders that I was able tt get after this where what I was expecting. By using both OSMnx and NetworkX I was able to highlight the path that the algorithm choose as well as the shortest path (Dijksra's Algorithm). The OSMnx render (Figure 4) shows the shortest path from point to point in red, and the optimal path according to the A*Prune algorithm in blue. As you can see, the blue path (A*Prune) avoids the middle of the map, while the red path (Dijkstra's Algorithm) goes right through the middle.

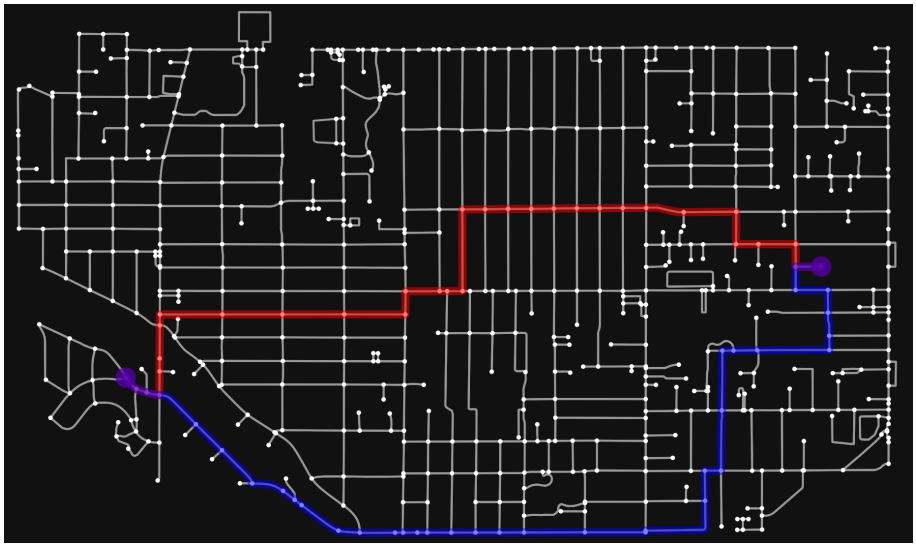

Figure 4: Dijkstra's Shortest Path, and A*Prune Path

Once I was getting results like Figure 4, I began to work on trying to add a heat map to overlay on top of the renders. I eventually was able to generate a heat map as seen in figure 5. The heat map represents the popularity of each edge, and since I was going for a city center looking heat map, the middle is the most popular with each color change signifying less populated edges.

Figure 5: City Center Heat Map for Brentwood - Darlington

When this heat map is then overlaid with the map of Brentwood - Darlington, it really shows how the algorithm is avoiding the middle (hot spot) of the map, and this what is suppose to happen (figure 6).

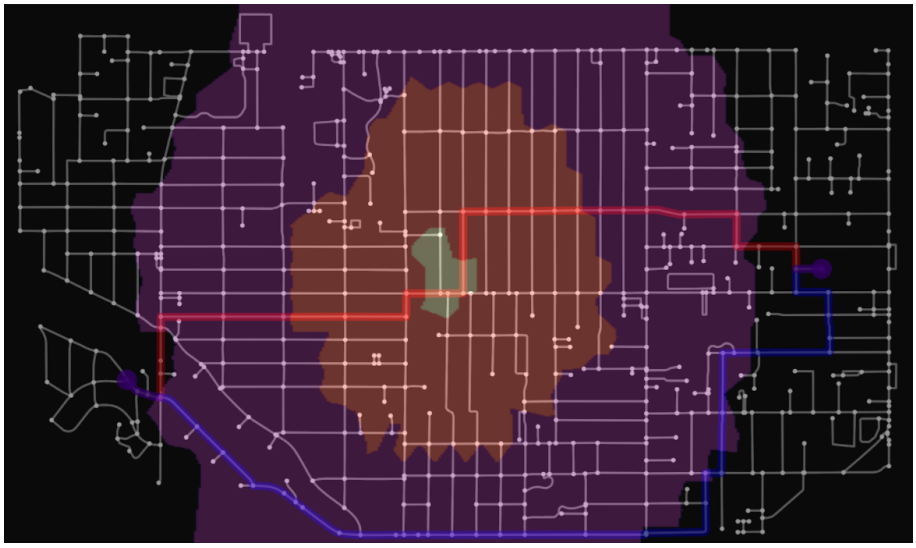

Figure 6: Brentwood - Darlington With Heat Map

Figure 6 shows how the shortest path (red) goes straight through the middle of the hot spot while the A*Prune path (blue) avoids the more popular paths and stays away from the middle minimizing popular paths/roads.

4 Results

After conducting tests, the results show that the A*Prune algorithm does indeed provide a path that is less popular then Dijkstra's shortest path. When looking into some of the numbers from the tests, it is seen that although the A*Prune path (blue) is longer then the shortest path (red), it has a lower popularity value then the shortest path (red). This can be observed in Figure 6. The total distance of A*Prune path (blue) is 15.6 percent longer then the shortest path (red) , but is 5 percent less popular then the shortest path (red).

Start Node	End Node	Dijkstra's Length (units)	Dijkstra's Popularity (units/length)	A*Prune Length(Units)	A*Prune Popularity (units/length)	%Longer	%Less Busy
89	225	31	3.35	35.83	3.18	15.58	5.07
500	200	39	3.15	46.05	2.97	18.08	5.71
510	421	36	3.25	44.75	2.93	24.31	9.85
1	112	12	6.66	18.14	6.34	51.17	4.8
94	427	23	3.17	31.93	3.38	38.83	-6.62
156	212	4	2	5.14	1.56	28.5	22
					Average	**29.41**	**6.8**

Figure 7: Results Table

The data in the table above (Figure 7) was taken form the Python code at each test. The percent longer and percent less busy columns were calculated with these equations. $percentlonger = ((A*Prunelength/Dijkstra'slength) - 1)*100$ and $percentlessbusy = ((Dijkstra'spopularity - A*PrunePopularity) / Dijkstra'spopularity) * 100$.

When looking at the distance and popularity differences for multiple trials, it is observed that the average A*Prune path (blue) is 29.41 percent longer, and 6.8 percent less popular then Dijkstra's shortest path (red).

An interesting result can bee seen in figure 7 on test 5 where the calculations show that Dijkstra's path is less busy then the A*Prune path. This is something that I am really unsure why it's happening.

5 Conclusion

Based on the results we can conclude that overall the A*Prune algorithm is able to find a path that is less popular then the shortest path, especially when the path goes thought the middle of the map where the hot spot is located. This causes the A*Prune path to divert around the middle of the map avoiding the "city center" hot spot.

Although the algorithm works for most staring and ending node combinations, there is still a lot of work that could be done with this project and other projects that are similar. This project could be worked on more to obtain even more desired results so that the A*Prune path is more customized and accurate. Another way this project could be improved, is for it to be a stand alone app or an app that can work directly with other workout tracking apps like Strava, or Garmin. This would allow the algorithm to tap into the data from these apps to provide a more effective route.

All in all this research project satisfies the project goals by providing its users with a way to get from one point to another that is less buss/popular then the faster, more busy route.

6 Acknowledgment

I would like to express my thanks to the National Science Foundation, Portland State University REU program, as well as my mentors Christof Teuscher and Philippe Proctor for all of there help and support.

References

[1] W. M. Carlyle, J. O. Royset, and R. Kevin Wood. Lagrangian relaxation and enumeration for solving constrained shortest-path problems. *Networks: an international journal*, 52(4):256–270, 2008.

[2] A. Deslandes, H. Moraes, C. Ferreira, H. Veiga, H. Silveira, R. Mouta, F. A. Pompeu, E. S. F. Coutinho, and J. Laks. Exercise and mental health: many reasons to move. *Neuropsychobiology*, 59(4):191–198, 2009.

[3] G. Liu and K. Ramakrishnan. A* prune: an algorithm for finding k shortest paths subject to multiple constraints. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2, pages 743–749. IEEE, 2001.

[4] E. Payne. A resource constrained shortest paths approach to reducing personal pollution exposure. *https://pdxscholar.library.pdx.edu/reu$_r$eports*/15/, 2019.