Optimizing Flights using multiple approaches

Muhammad Abdurrehman Asif

CS110 Fall 2020

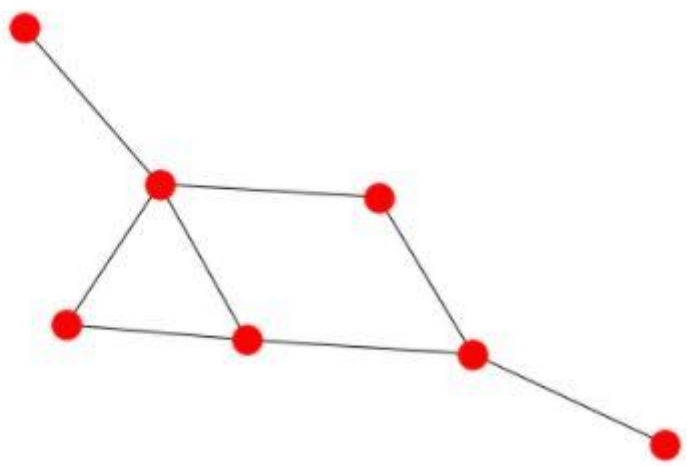
Abstract

As the complexity and connectedness of the various networks in the world increases, it only poses the question of how to optimize any functionality within these networks. We see relevant networks in all parts of our lives nowadays. From Amazon creating a spiral web of efficient delivery systems, college students indulging in sharing the latest news and gossip to airports bringing distant locations within the reach of eager travelers. In this paper, we will be looking at ways flights can be optimized. This is a very relevant problem as with the advent of air travel, there is an exponential increase in the number of planes in the sky. Averagely, at any given moment, there are approximately 5000 aircrafts in the sky (FlightRadar, 2020).

The Basic Problem

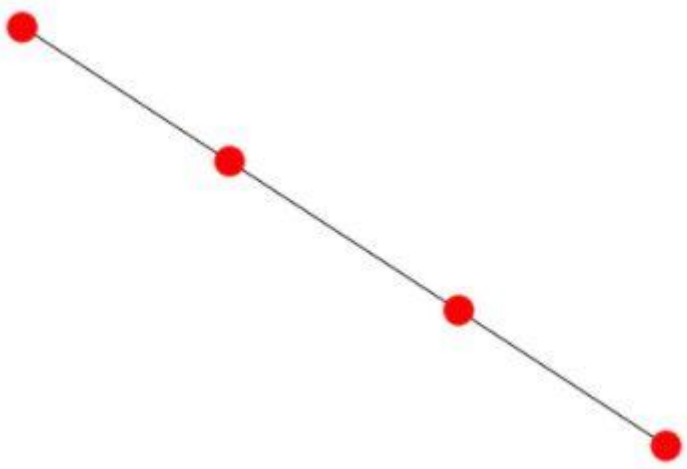
At first glance, the problem seems somewhat arbitrary. It seems simple enough for airline carriers to look at two locations and judge whether it is feasible to operate between these ports. This makes things even simpler for customers, who can just prioritize selecting direct flights. For example, if I wanted to go to New York from San Francisco, I could go to the front desk of my preferred airline counter (let's assume for now that it's Delta) and just book a direct flight or the shortest one available.

The problem that we just outlined is simplified to the level where we can start applying logical tools to tackle this. First and foremost, we need to understand what this problem is on a skeletal level. What it says is that there can be N= number of locations, which may be directly or indirectly connected. These locations mimic a 'network' and certain clustering may form as well.



A simple Python generated structure showcasing a microscope of what this may look like.

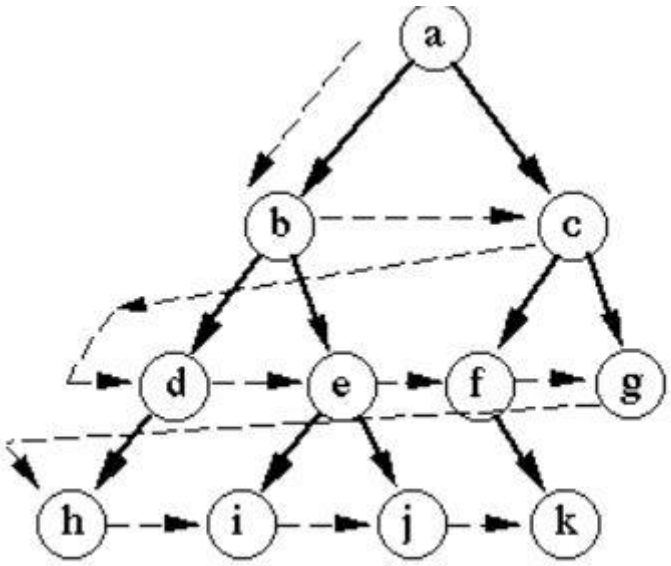
We wish to get from location A to location B. Now these locations can be connected by a subsequent set of nodes in the following way.



It is important to note that currently these nodes do not represent anything or hold any sort of value. We are just creating the visual outline of what our problem seems to represent. Taking the above diagram as an example, to get from the left most node to the right most one, we have to travel between 2 locations. This is exactly what our basic problems solution is. In a densely populated network, identifying the appropriate connectors and connections is what will get us to our required path.

A Breadth First Approach:

A breadth first search(BFS) algorithm is one that traverses a suitable data set in order of breadth. Meaning, that once it arrives at a node, it looks at all the possible options it can go to and chooses the most optimal one based on that, and then does the same over and over til it finds a solution. If we were to visualize this search, this is what it would look like.



The breadth first approach provides some great benefits

1. It is 'pseudo'-greedy. Meaning that it looks for optimal sub-solutions at each level but does not eliminate options.
2. It verifies each and every single possibility.
3. Through a densely populated network which is hard to understand, it can uncover overlooked links or routes.

But it also has its drawbacks. First of all, a breadth first search is very computationally expensive. It has to go through each node and then each edge. This can have a time complexity of O(|V|+|E|) in which V is the total number of nodes and E is the total number of edges. For very large data sets, such as real life airport connections, this can take a very long time to process. If for instance, someone is trying to look up these flights at the airport counter as discussed previously, all this waiting can lead to a long line accumulating.

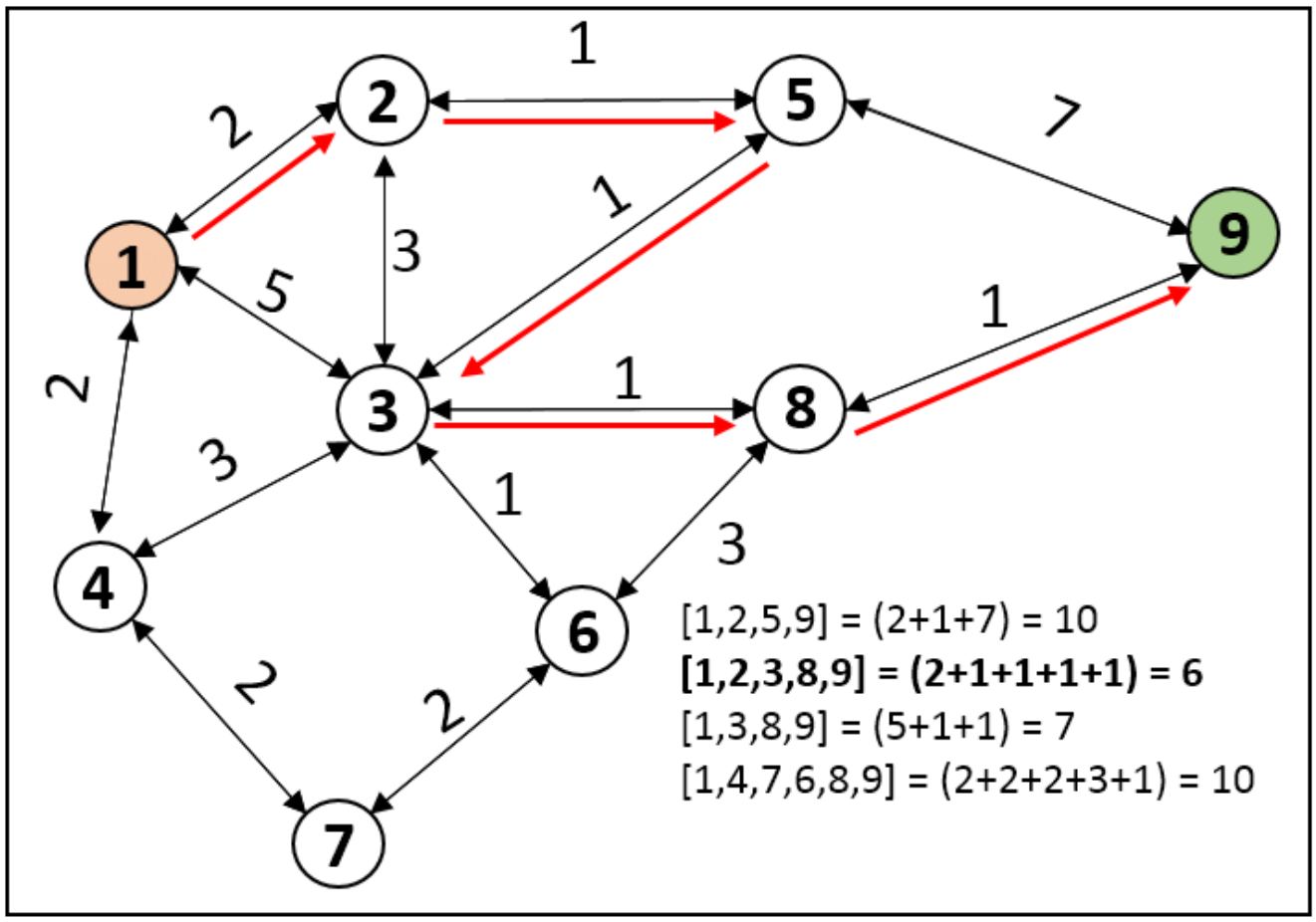
Another issue, and perhaps the key one is that the BFS cannot incorporate an additional layer of nuance which is so relevant in such a problem. Finding a 'truly-optimal' flight path has to take into account more than just what connections are available. Flight price, stopover timing, carrier of choice, take-off and landing times are all valuable things that people consider. In a simple Breadth-First Search, we do not have any weights for the edges. Each and every connection between two vertices is weighted the same, meaning that there is no difference between going from point A to B and going from point D to C. The BFS, therefore can be called a 'shortest flight mapper'. This is because it looks for direct connections that lead from point A til B. Direct connections can reasonably be assumed to take shorter than the same distance broken up over N legs.

The Complicated Problem

The extra layer of information that will be included in this problem revolves around price optimality. Price optimality is something that can constantly predict similar trends of pricing between two airports. It is given on a scale from 0 to 15, with a lower score indicating more optimality. This follows simple logic. Price optimality changes when a certain flight is going to or from a 'hub'. For example, JFK airport and Philadelphia international airport are located in relatively close proximity. A flight from JFK to LAX *should* be more price-optimal than a flight from Philadelphia International to LAX. The reason is that JFK is an air traffic hub. More flights, more carriers and more passengers travel from it, meaning that the increased demand for flying to locations is greater. Another example would be flying to JFK versus flying to Portland Intl Jetport (Located in Maine). Both these airports are separated by just over an hour of plane travel, however, *flying to* JFK should be more price-optimal than flying to PIJ for similar reasons as stated above.

Dijkstra's Algorithm Approach:

Dijkstra's algorithm is an effective algorithm to tackle the complication that arises with the problem. This is because Dijkstra's algorithm can implement weights on edges. It follows greedy algorithm principles in which it looks for optimal subsolutions starting from our current location. Each node has a weight of infinity acting as a sentinel and in an iterative process, all possible routes from the start are looked at until the minimum valued edge is found. Then the node that is connected to the edge's infinity has it's value replaced to the value of the edge. Further exploration occurs from that node until all routes to the destination have been mapped and each node value has been replaced by the minimum value of the node plus edge weight.



This figure clearly showcases how even when the possibility of visiting fewer nodes til the destination exists, there are more optimal ones which require an extra traversal.

Dijkstra's algorithm takes a time complexity of O(V^2)where V is the total number of vertices, or O($V + ElogV$)where E is the total number of edges. This distinction depends on the implementation and it will be discussed in the algorithm comparison part of the paper later on.

There are two clear disadvantages of using this algorithm. First of all, since the algorithm follows similar traversal methods, it will also be time inefficient. This inefficiency can be reduced of course and implementation techniques can make this better, but just the traverse-find-compare-replace aspect of the functionality means that it will take time.
Secondly, Dijkstra's algorithm can not incorporate negative edges. This is extremely pertinent to the discussion of greedy and dynamic algorithm implementation. Dijkstra's algorithm functions on the logic that local optimums combine to create global optimums. This means that adding a node in the order of the destination when it has already been established that the current node is at minimum will *ALWAYS* 'add on' total weight. If we have established an optimal direction, then all nodes in that sequence will 'add' value to our total, and same is the case for the sequence of nodes we are not traversing. By adding negative weights, this contradicts this logic and therefore the foundation of the algorithm itself. Adding an edge can never make paths shorter. If negative edges existed, then our algorithm might be lured into a small-valued edge and go down the path of ever increasing paths, while there being a route in which an expensive edge is deemed sub-optimal but the following sequences may lead to significantly smaller paths.

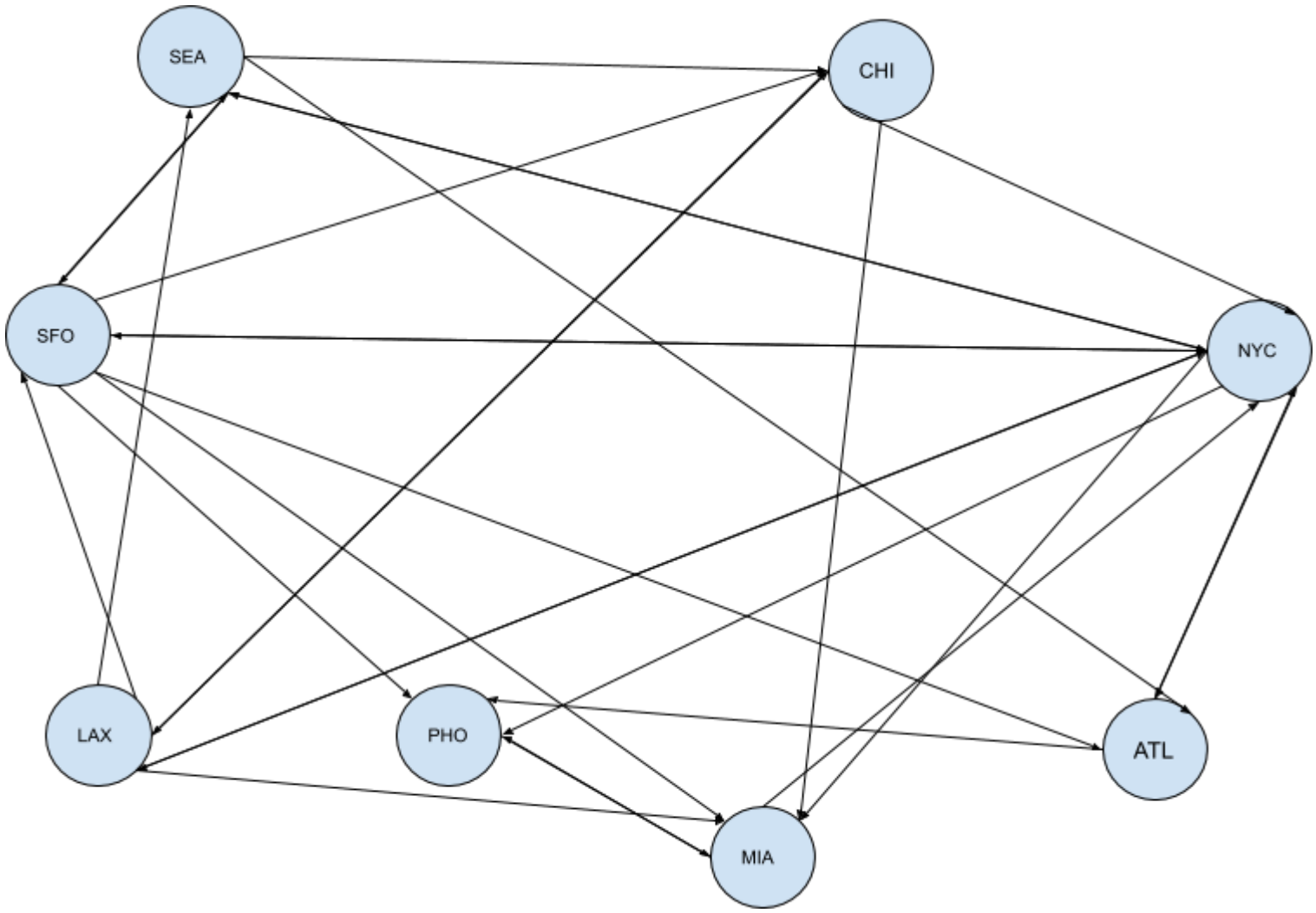
Implementation of BFS:

My implementation of the BFS algorithm relies on using a graph. The reason why this works is due to adjacency lists. A graph can map out adjacent vertices and connect all the possible data points in a convenient network. This network structure is easily understood and used by python making us think of each step of the algorithm in an intuitive way.

The depth first algorithm I have implemented works in an iterative process. It starts by taking a graph, a starting point and a destination as an input. From the starting location, it starts looking at neighboring vertices. A while loop runs with the base case being that we have reached our destination node. If the currently explored node is not our destination, the algorithm goes back and looks at all possible neighbors. Even then if the node isn't found, it moves onto the adjacent vertices and continues searching. As mentioned before, the runtime of this algorithm is O($|V| + |E|$) which makes it somewhat inefficient. It also does not have to care for any weights on any edges as all edges are unweighted and therefore all nodes are unweighted.

I have used a total of 8 US airports which mimic the sort of relationships we might observe in real life. Actually implementing this nationwide would require a thorough dataset, one I did not have access to and is beyond the scope of this paper. There are 2 ‘hub’ airports I have selected: New York City and San Francisco. Los Angeles is a close second and that choice was intentional to make meaningful comparisons in connections from LAX versus SFO, since they are relatively close.

The graph below shows this relationship.



The eight selected cities and airports are: Seattle, San Francisco, Los Angeles, Phoenix, Miami, Atlanta, New York and Chicago. The arrows indicate where the flight leaves and where it lands. Double-sided arrows indicate that connections exist both ways. The diagram above tries to generally map the location of these cities on the US map, with NYC being the east coast hub and SFO being the west coast hub.

The map above shows how there are countless ways to get to from one point to another. For example, to get from Miami to Phoenix, we could either fly directly, or we could go through New York. We could even take a longer route by going from MIA → NYC → SFO → PHO, or if we want to fly west coast then MIA → LAX → SFO → PHO. However, the BFS algorithm will always return the shortest flight it finds, meaning the least amount of connections as possible.

Implementation of Dijkstra’s Algorithm:

My Dijkstra implementation takes use of some key ingredients.

1. First of all, I have used a graph data structure here. The rationale behind this is simple. Using a graph provides me the opportunity to play around with the traversal as if it was a tree. If I had used arrays or lists, it would have meant that the algorithmic efficiency would take a large hit. Adjacency list representation grows exponentially once we are performing searches similar to breadth first.
2. Setting up the algorithm to have sentinel values at each node is what makes sure that the ‘lowest possible’ combination of edge weight and node value is ultimately stored. This is what looks for local optimals and combines them to find global ones. If sentinels were not set, the algorithm wouldn’t be able to decipher between two routes. Even initializing the graph with ‘0’ as the node value would mean it is more optimal than any travel negating our logic.
3. I have used min heap and priority queues in this algorithm. The priority queue is the ‘price optimality’ and the min heap pops the root giving us the most optimal path. This is a choice made for efficiency as well. Seeing as the graph could be traversed in a tree form, using heaps and pqueues made the most logical sense.

4. The approach I have used is an iterative one as opposed to a recursive one. Even though recursion may be useful while implementing this, the greedy nature of the algorithm allows us to make locally optimal decisions at each level of traversal. Since we have established that due to Dijkstra's algorithmic logic, the local optimals combine to get the global one, an iterative approach will work just as effectively.

Although it would be very helpful to visualize the same graph as done in BFS with the weights attached, due to the graph containing 29 connections spread over 8 airports, it would look very messy and muddled up. Therefore, this table below shows the relationships and weights

Starting Airport	Destination 1	Destination 2	Destination 3	Destination 4	Destination 5	Destination 6
LAX	SFO: 3	SEA: 4	CHI: 10	NYC: 6	MIA: 5	
SFO	SEA: 2	ATL: 7	CHI: 8	NYC: 6	MIA: 4	PHO: 3
SEA	ATL: 10	CHI: 5	SFO: 4	NYC: 6		
CHI	NYC: 4	MIA: 9	LAX: 2			
NYC	MIA: 3	PHO: 2	LAX: 5	SFO: 4	SEA: 5	ATL:2
ATL	NYC: 2	PHO: 5				
MIA	PHO: 8	NYC: 4				
PHO	MIA: 9					

As we can see, there were some intentional decisions behind the scoring of the weight. The weights are however arbitrary and are not backed by any form of evidence or data. This application and implementation is purely theoretical, however, I have tried to make this theory as realistic as possible. Flights to and from hub airports should be more 'price-optimal'. Low operating airports such as Miami and Phoenix should have less 'price-optimal' flights.

Results and Comparisons:

Output

The optimality conditions that were inputted into Dijkstra's algorithm resulted in results that varied from the BFS. The BFS algorithm is efficient if we are looking for direct flights. We could also call this a 'shortest absolute flight' algorithm; absolute because we are not considering time spent on layovers. However, with the inclusion of price optimality, the output was different. For example, while trying to find the best route from Miami to Phoenix, the BFS algorithm suggested a direct flight, however, Dijkstra's algorithm suggested we go through NYC. This can be attributed to a cheaper connection overall as the price optimality score for a direct flight from MIA to PHO is 8, but one including a stopover at NYC is 6. Another great example is Chicago to Miami. There is a direct connection available, however, flying through LAX results in a more price-optimal route.

Time Complexity

As mentioned above, the time complexity of BFS is O(|V|+|E|). This is not really suitable. It can be considered inefficient but that is expected since that is the nature of the program. The Dijkstra's Algorithm that has been used has a runtime efficiency of $O(V^2)$. This is worse and it means that for extremely large input graphs, the algorithm will take very long to run. This can also be pinned down to the way that the algorithm is designed. If we were not tracking nodes, pointers and weight changes, this could be cut down. If we implemented a fibonacci heap and an adjacency list, the runtime could be cut down to $O(V + ElogV)$, although this does not beat the simple BFS either.

Another strategy is to perform termination after local optimals have been found. All vertices that have been traversed at that level and are not equal to or less than the local suboptimal can be terminated at each stage. This simplifies the tree and gets rid of routes that won't be used. The tradeoff here is that if there are shorter routes later on which are prematurely terminated, our solution would be inaccurate.

Space Complexity

BFS searches take O(|V|) memory to store all the vertices. Since it does not remove or store any variables at different times of the algorithm, this property is maintained. Dijkstra has the same complexity. This is because it too has to only store all

the nodes that exist in the graph. No temporary lists or adjustments are made. However, there are certain constants we should not ignore while analyzing Dijkstra. For instance, all the edges and weights will be changed while selecting a start and finish. Depending on the size of the graph and the number of edges, these can be significant leading to a big chunk of the memory.

Further uses

Currently, only the commercial uses have been looked at for the scope of this assignment. However, air travel is used for so many more purposes. One of these is the transfer of cargo. Since the BFS and Dijkstra's algorithm both find shortest and optimal routes respectively, we can also look at stripping the graph away to create a basic outline for how cargo can be sent all across the country. For this, we will have to use a '*minimum spanning tree'*.

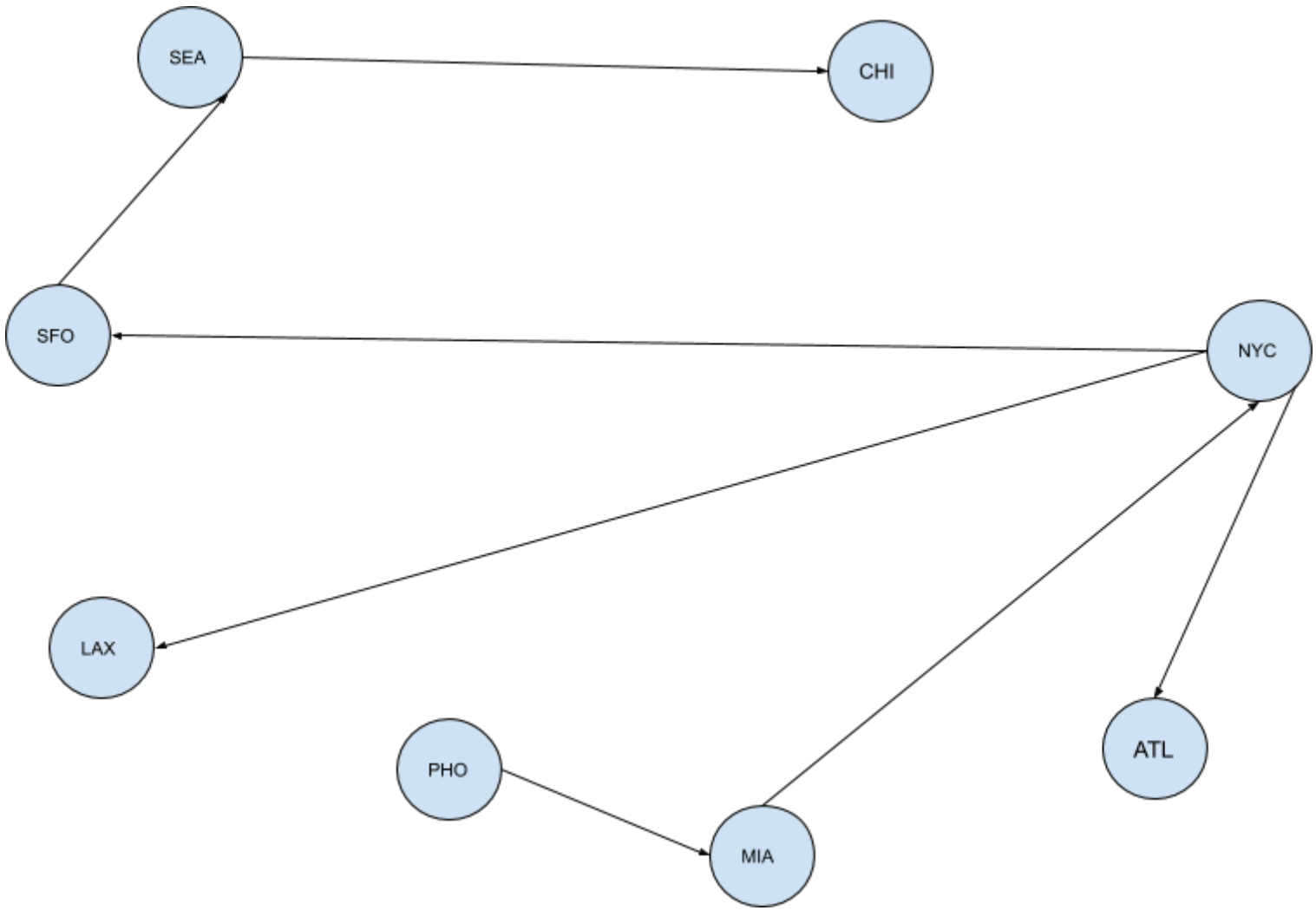
A minimum spanning tree or an MST is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight (Wikipedia). We can create an MST from our graph to see how cargo companies can set up basic travel routes to get to each corner of the country. We can pick an arbitrary starting point, and then from that we look towards the shortest weighted path. Once we find this, we find another short path which includes new nodes. We do this across the graph ensuring that no cycles are created. The algorithm we are going to base the implementation off of is Prim's algorithm. Prim's algorithm is used to create minimum spanning trees with no cycles by creating two sets of vertices, one containing visited and the other containing non-visited nodes. After this, at every step where the two sets connect, the lowest weighted edge connection is chosen and this continues until all the nodes are connected. It is also a greedy algorithm as it looks for local optimals that combine to form global optimals and has a runtime efficiency of O(V^2), V being the number of vertices. However, by using Fibonacci heaps the runtime can be improved to O(E + logV).

By running the MST algorithm, this is the output we get:

In [9]:
```
def create_mst(graph, start):
    mst = defaultdict(set)
    visited = set([start])
    edges = [
        (weight, start, to)
        for to, weight in graph[start].items()
    ]
    heapq.heapify(edges)

    while edges:
        weight, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst[frm].add(to)
            for adjacent, weight in graph[to].items():
                if adjacent not in visited:
                    heapq.heappush(edges, (weight, to, adjacent))

    return mst

flights = {
    'LAX':{'SFO':3, 'SEA': 4,'CHI':10, 'NYC':6, 'MIA':5},
'SFO':{'SEA':2, 'ATL':7, 'CHI':8, 'NYC':6, 'MIA':4, 'PHO':3},
'SEA':{'ATL':10, 'CHI':5, 'SFO':4, 'NYC':6},
'CHI':{'NYC':4,'MIA':3, 'LAX':6},
'NYC':{'MIA':3,'PHO':2, 'LAX':5, 'SFO':4, 'SEA':5, 'ATL':2},
'ATL':{'NYC':2,'PHO':5},
'MIA':{'PHO':8, 'NYC':4},
'PHO':{'MIA':9}

}

dict(create_mst(flights, 'PHO'))
```

Out[9]:
```
{'PHO': {'MIA'},
 'MIA': {'NYC'},
 'NYC': {'ATL', 'LAX', 'SFO'},
 'SFO': {'SEA'},
 'SEA': {'CHI'}}
```

Graphically, this can be represented as:



This shows that a cargo company based in Phoenix can reach any of these 8 cities through these base connections. Repeating this with any other input city will also give us a MST which can be used for similar purposes.

Conclusion:

In conclusion, multiple different approaches for the same problem can be used. It is largely dependent on the end-user, what sort of outputs they prefer, and so using a variety of methods can be eye opening in terms of what different algorithms can offer. For example, an app like Expedia may prefer Dijkstra's algorithm to provide optimal combinations of price plus flight to their customers. Specific Carrier applications such as Delta's app may prefer a BFS algorithm for their reward customers. Customers in high-tiers such as ones that travel exclusively in Business Class and above may not find pricing the deciding factor in their choice. Their prioritization of maximum comfort can lead to flight apps such as the Delta App recommending the least stopover flight. Finally, warehousing companies such as the ones partnered with Amazon may wish to use Prim's Algorithm to see how their port can be connected to all possible destinations. Of course, there are downsides to using these algorithms as we have seen them take very long as the inputs grow exponentially. Finding a way to combine all of these into a dynamic search engine which allows the user to select filters (price, number of stopovers, carrier, etc) can make this a very valuable program.

HC and LO Application:

#GreedyAlgorithms: I used two greedy algorithms to complete this assignment. Dijkstra and Prim's algorithm are both greedy. I outlined what makes them greedy, what aspects are beneficial and what aspects can be improved due to their nature. The computational analysis was directly tied to the greedy nature of the algorithms which was key to finding the solution to the problem. I haven't had a chance to apply this LO many times in class since it came at the latter part of it, but I have enjoyed the introduction to greedy and dynamic programming because they provide a more intuitive way to solve problems, even if it is harder to write in Pythonic language.

#ComputationalSolution: I chose this LO because it encompasses more of what my assignment is about. The assignment is about solving a problem creatively and trying to approach it from multiple perspectives. Analyzing, comparing and contrasting these approaches was the strongest application of this LO and finally finding ways to build onto the solution was what I enjoyed the most. Over the semester, I have grown more comfortable with applying computational thinking to problem solving. From all

the data structures to all the types of algorithms that can be created, I think the biggest takeaway for me is finding the right fit. A way to solve a problem that is computationally efficient and algorithmically simple is the optimal thing to search for.

#PythonProgramming: I started working on this project about 3 weeks ago. My coding skills were never the best, and I never particularly enjoyed coding. Even though I was doing coding for pre class works, there were still pseudo codes and peers available who helped me. When I realized that I needed to learn coding and wanted to find ways to enjoy it, I started doing projects and wanted to do this one for my final. I learned Dijkstra's algorithms logic, implementation and coded it along with BFS. After this, when I felt more comfortable, I tried coding Prim's algorithm and with little to no help I was able to come up with the algorithm for it.

#CodeReadibility: Since I knew I was trying to learn as much as possible, and that I would be working on this for a long period. I tried writing code that was simple to read, simple to understand and well documented. I wanted to write something that not only I could understand, but anyone who was trying to learn about Dijkstra, BFS, MST or even graph data structures could read and understand. Over the semester, I think I have been consistent with his LO since I always try to make code as readable for myself to go back to.

#**DataStructures:** The biggest challenge in this assignment was learning about a completely new data structure: graphs. Moreover, learning about the functionality possible with graphs, their implementation and usage. I tried three different computations using graphs and by the end got very comfortable with how graphs worked. Over the assignment, I have mentioned reasons for why graphs are suitable, what properties they have, what functions we can perform on them and how complexity of the algorithms I wrote are influenced by them. Over the semester, I have gotten to learn about so many data structures. The biggest learning for me isn't that I learned all these data structures, rather, I learned ways to think about data structures themselves to understand them. I learned key questions to ask when faced with a new structure. I learned how to look for answers, how to look for pseudo code and where to start understanding these. This was my greatest learning from CS110.

#Optimization: This assignment was about optimization and I constantly was thinking from an optimization point of view. Even when I had answers, I was thinking about how to improve, how to build upon what I have. I started with BFS, improved optimization using Dijkstra and then created an MST for additional functionality. Therefore, I provided multiple lenses of optimization.

An .ipynb file containing the code will be attached to the assignment, but the code can be accessed using this link as well.

References

GeeksForGeeks. (2020, November 03). Prim's Minimum Spanning Tree (MST): Greedy Algo-5. Retrieved December 18, 2020, from https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

Python-Course. (2020). Python Advanced Course Topics. Retrieved December 18, 2020, from https://www.python-course.eu/networkx.php

Sawhney, D. (2014, December 24). Why doesn't Dijkstra work with negative weight graphs? Retrieved from https://www.quora.com/Why-doesnt-Dijkstra-work-with-negative-weight-graphs

Wikipedia. (2020, December 07). Dijkstra's algorithm. Retrieved December 18, 2020, from https://en.wikipedia.org/wiki/Dijkstra's_algorithm

Wikipedia. (2020, December 07). Breadth-first search. Retrieved December 18, 2020, from https://en.wikipedia.org/wiki/Breadth-first_search