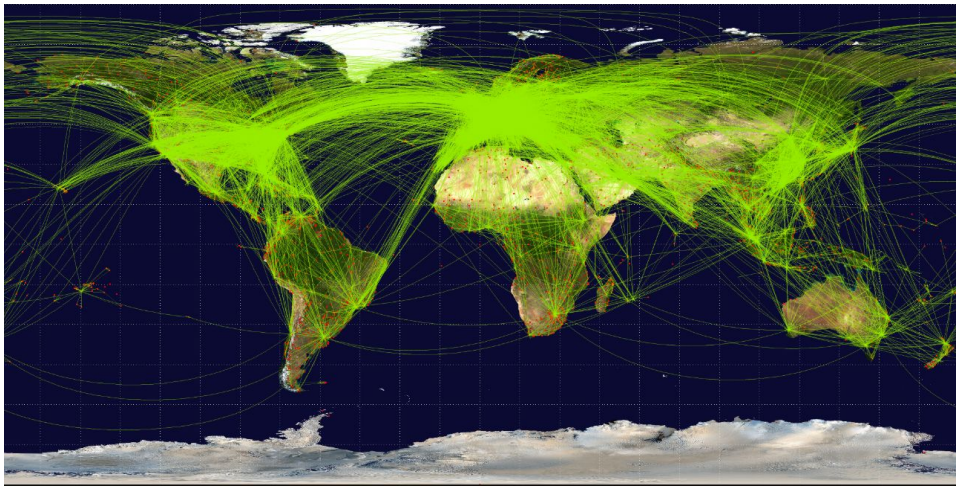
Under the Radar: Alternative Fare Comparison

Tom Kremer

CS110 Fall 2018



Introduction

Flight fare-comparison services are extremely popular web applications, particularly among students who travel every four months to a different side of the globe. These students (and many others) are often interested in finding the cheapest way to get from A to B even if it requires additional stops. However, based on my experience, many popular services like Skyscanner are not comprehensive in their coverage of alternative routes in the name of savings. To stick with that example, Skyscanner presents an "Everywhere" feature showing cheap flights options from a source airport, but they do not fully use it when suggesting paths for standard searches — many times I found cheaper fares by manually iterating through the cheap options in the "Everywhere" list and looking for flights from them to my destination. This was the motivation behind this application, titled (unoriginally, I admit) "Under the Radar".

The application holds a database of ~3000 airports and ~36000 unique routes between them, each containing the cheapest available fare for the route. The airport and paths (and their accompanying visuals, featured in the cover) were retrieved from OpenFlights organization, while fare prices were generated randomly in absence of publicly available data, with domestic flights having a lower range of possible prices than international.[1] The principle behind the algorithm is not impacted by made-up prices. While I did not include factors like cancellation or delay rates, introducing them to get a more realistic model (if I could later-on obtain real prices) is simple, only requiring to decide on the weight they will play in affecting the fare, without affecting the algorithm itself. The current application allows the user to insert source and destination airports and a maximum number of stops and returns the cheapest path existing.

In the rest of the paper I present my data structures and algorithm choices, introduce Dijkstra's Algorithm and the optimizations I made to suit it for this application, analyze my implementation's time and memory asymptotic and empirical behavior and present a brief overview of my code's structure.

[1] A correspondence with ATFCO, the "standard setter in the airfare ecosystem", in order to obtain real flight data is still undergoing.

Data Structures and Algorithm Choices

The types of data used in the application are airports and routes. Airport data, including its 3-letter IATA code serving as the airport identifier throughout the application, name and location are stored in a Python dictionary indexed by IATA code. Route data is stored in a graph made of Airport objects serving as vertices, each hold an 'adjacency' Python dictionary in the form ('APT': price), APT standing for each neighbor airport and price for the cheapest fare price between them. To include additional factors, I could simply modify the price as I wished (e.g. multiply by the delay rate). While each Airport object could also theoretically hold a "Description" attribute with the data currently held by the Airports dictionary, I found it more convenient to separate them as my Dijkstra's Algorithm uses the Airport object as a whole (which might take unnecessary memory if also holding a long string), and in an extended version more Airport data could have been needed, justifying a separate form of storage.

I implemented an adapted version of Dijkstra's Algorithm to find the cheapest route between two given airports up to a given number of stops. I chose this algorithm because the problem can be represented by a graph in which airports are vertices, direct flights between neighboring airports are edges, and prices are weights. Since there are only positive integer weights, Dijkstra's algorithm provides the appropriate approach, as the shortest weighted distance between two nodes is the cheapest flight path connecting them.

Dijkstra's Algorithm is a greedy algorithm that finds the single-source shortest path to all other vertices or a target vertex in a graph. In its original version, each vertex is given a measure of distance from the source vertex *s* initialized at *d=0* for the source and $d = \infty$ for all others, and a pointer to the previous node or path to the node. All vertices are then inserted into a data structure to be iterated on, e.g. a queue. The algorithm runs until the queue is empty, extracting the node *u* with the lowest distance in each iteration (hence the greedy property), and 'relaxes' each of its connected vertices, hereafter '*v*'. Relaxation consists of checking whether reaching *v* through *u* results in a shorter path from the source than previously found for *v*, i.e. $if\ u.d + weight(u,v) < v.d.$ If so, it updates *v*'s shortest path such that $v.d = u.d + w(u,v)$, changing its position in the queue, and adds *u* to *v*'s path list. Without a target node, when the queue is empty, all nodes were processed exactly once, each reachable node from *s* has the

shortest weighted distance from *s* as *v.d*, and infinity if unreachable. With a target *t*, the algorithm can be stopped when *t* is extracted from the queue because when vertex *u* is extracted from the queue *u.d* is the shortest distance from *s* for this vertex. A simplified reason for that is that once *u* is extracted it has the minimal distance from *s* out of all unvisited vertices by definition (the greedy property mentioned earlier), which means there is no other vertex with a shorter distance from *s* through which a shorter path to *u* can be created (Cormen et. al, 2009).

In my implementation, I used the *heapq* Python module to create a min-heap queue, in which insertions and deletions take O(*log* queue size) time. I used two Python dictionaries to hold the distances and paths of each node. Holding the full path helped to define a condition such that if a new path is equal to the existing cheapest path, $v.d = u.d + price(u \rightarrow v)$, I kept the route with least stops, i.e. shorter existing path. The elements in the queue are tuples in the form of (*price*, *Airport*), allowing easy access to each airport's route list. Having a target airport, I stopped the algorithm after it is extracted from the queue.

I made several adaptations to the standard algorithm to save unnecessary relaxations, i.e heap operations, made possible under the application's constraints. First, since the user provides a maximum path length, I prevented the algorithm from relaxing longer paths that lead to more stops than allowed. I differentiated between an airport one step before the limit, for which the algorithm only checked if a flight to *t* exist and relaxed it if so, and an airport with less than one step before for which I relaxed all neighbor-airports. This way, only relevant path were examined and inserted to the queue. Second, in the relaxation process, in addition to the standard $v.d > u.d + price(u \rightarrow v)$ test, the algorithm also compares the new path cost, $u.d + price(u \rightarrow v)$ to *t.d,* the existing shortest path to *t.* If it is already more expensive than *v.t*, it couldn't generate the shortest path to *t*, so there is no point in relaxing it and pursuing a path to *t*. If *t* was not found yet ($t.d = \infty$) any path would always satisfy this condition, and if the examined airport is *t* it just duplicates the regular check for *v.d*. Finally, before starting to iterate through the airports, the algorithm checks whether a direct flight exists between *s* and *t* and if so, it adds the relevant (price, *t*) tuple to the queue. This provides an early upper boundary that also assists the previous addition. These three optimizations save precious time by reducing the number of paths

examined, i.e. heap operations performed — the only varying element in the algorithm's running time, as I discuss later on.

Another modification I made was subtituting the decreasing of keys of queued nodes by pushing a new instance of the node with the updated (lower) price. This is still correct as only the cheapest one will be extracted and returned, but increases the queue size and amount of potential queue operations, and was mostly a choice for simplicity. As a minor compensation, I did not push all airports to the queue upon initialization, but only when upon successful relaxation. Eventually, these slight modifications to the queue size were negligible due to the effect of the optimizations mentioned earlier. All heap operations, in my case push and pop, are logarithmic in the queue size, and the maximum queue size observed in simulated runs with up to 14 stops (who the hell would search for that) never even passed V, the number of airports (~3100), with maximum values of about 2000 (figure attached in the next section). The logs of these numbers are very similar. The low queue size can be attributed to the previous optimizations, and only positive weights mean longer stop caps stop yielding valid cheaper paths at some point, as shown in the next section.

Asymptotic Time Complexity and Memory Analysis

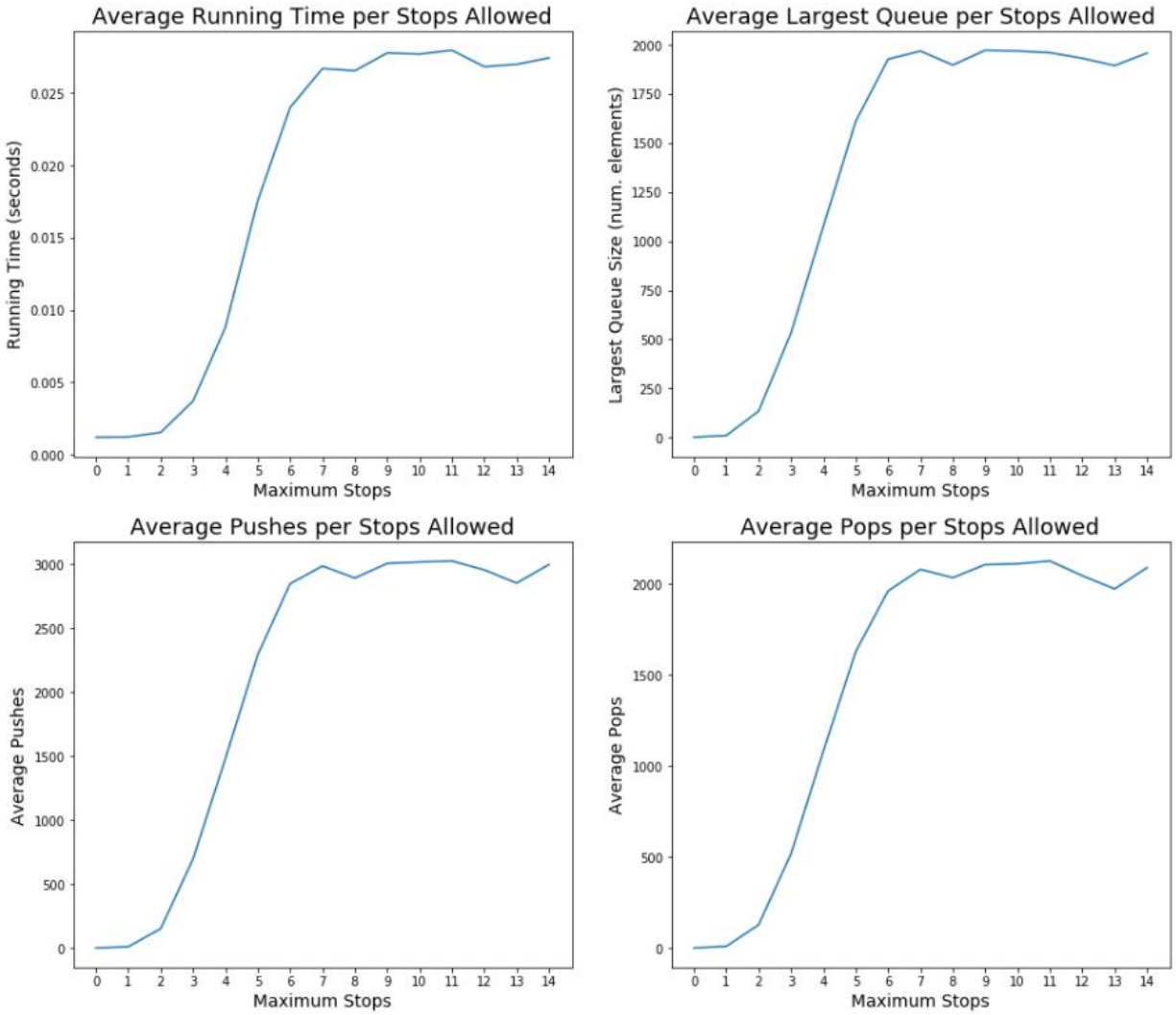
Let V be the number of airports and E number of routes, which in my case are about 3000 and 36000. The asymptotic time complexity of Dijkstra's Algorithm depends on the number of queue operations performed and their complexity. Initialization of distances and paths for each airport takes 2*O(V)=O(V) time, and the added check for a direct flight takes constant time. In my case, there are only pushes and pops, each taking O(log(queue size)). Empirically, the queue size never passed V, and was on average less than half of it, but in asymptotic terms, I will stick to O(log E) per operation, since at the (very) worst case the queue could hold all elements. The number of pushes is at most E, if every edge on the graph is explored. However, since paths longer than allowed stops and more expensive than the current cheapest are not pursued, the number of inserts is practically significantly lower. Similarly, the number of pops is bound by the number of insertions. In the worst case, all paths are pushed and popped, meaning the asymptotic complexity of my implementation is $O(Elog(E) + Elog(E)) = O(ElogE)$. In

practice, as shown in the next section, the number of pushes and pops even for as much as 14 stops is lower than V.

In terms of memory, the algorithm uses two Python dictionaries with length V keyed by strings and holding integers (price) and strings (path), and a queue that could reach size E at most but practically is smaller than V. The queue elements are tuples of price and Airport objects.

Practical Performance

I examined four measurements of performance: running time, maximum queue size (affecting each heap operation's time and the memory occupation), number of pushes and number of pops. As established so far, the performance of the algorithm is mostly influenced by the allowed number of stops, making it the dependent variable. For each number of stops from 0 to 14, I simulated the algorithm 300 times with random source and target airports to account for the diversity of airport connectivity. The results are presented in the following figure.



All measures scale similarly with respect to the stops, first increasing and then stabilizing around seven stops. This makes sense as decreasing additional paths that satisfy the conditions for a potential cheapest flight are likely to be found as additional flight prices are added with every stop. The running time of the algorithm (upper-left) is consistently fast with less than a tenth of a second. The average maximum queue size (upper-right) is lower than V, guaranteeing fast heap operations. The number of pushes (lower-left) also stays below V, and the number of pops (lower-right) is therefore also low, thanks to the optimizations allowed by restricting the maximum path size.

Code Structure

The code is attached as a Jupyter notebook along with the relevant CSV files containing the data. The code is divided to several sections as follows: Creating the Airport and Graph classes; Importing airport and routes data from CSV files and populate the relevant data structures; The adapted Dijkstra's Algorithm to find the cheapest route up to a given stop limit; Auxiliary functions to get user input and print output; Code for the Performance Analysis section including simulation and graphing; Main section to run the application; Testing zone with a commented simple graph and ready-made specific calls to Dijkstra.

Bibliography

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* Cambridge, MA: MIT Press.

Airport, airline and route data (n.d.). *OpenFlights.* Retrieved from: https://openflights.org/data.html