Heuristic Learning with Triplet Network for Flight Interlining Path-finding

Man Wai Kwok

University of California

Riverside

Student ID: <removed>

Master of Science in Engineering, Data Science specialization

Author Note

Email: mkwok012@ucr.edu

Code available for review: <removed>

Abstract

Flight interlining was drawing more attentions for its huge market potential by allowing freely combining flights to form a transfer itinerary while customers could enjoy services exclusive to a non-stop flight. Effectively suggesting valid, and even personalized, combinations became a technical problem to solve. This paper presented a heuristic model based on a triplet network, which was trained to learn from generated departure-transferring-arrival (DTA) journeys to make suggestions on transferring airport given a pair of departure and arrival airports. The model achieved a 90.3% of averaged recall rate, and with a tolerance of 5 airports, the recall rate could even reach 98.8%. Ordering of the suggestions was examined with the nDCG (normalized discounted cumulative gain) score. While traditional triplet loss did not take the ordering into account, it was found that improvement was possible with a modified triplet loss. Results also showed that when taking 10% of the data as the test data, a model trained on the other 90% could achieve a 63.5% of average recall rate on the test dataset, indicating some degree of predictability.

Heuristic Learning with Triplet Network for Flight Interlining Path-finding

Interlining agreement allowed two airlines to cooperate a flight itinerary composed of two or more legs, so that customers could travel to their destinations in multiple flights while enjoying services exclusive to single flight journeys, such as without the need to care about luggage transfer between flights. However, such choices were limited by available signed agreements.

To unleash the potential of interlining, the International Air Transport Association, or IATA, proposed a new framework (Coles, 2019) which would let any airlines, including low-fare ones, to join. The framework would be like a product catalog such that scheduled flights from the joining airlines might be combined to form a brand new or a hot itinerary, and through providing ancillary products and services, this could be a new market worth $3.3 billion.

To enable interlining, or equivalently combining legs, a path-finding algorithm which could effectively and efficiently select a transfer location given a departure and an arriving locations became highly relevant. However, the complexity of such problem grew with the number of flights and airports.

Based on a year 2013 dataset by OAG Aviation Worldwide LLC (2015), roughly 3,200 airlines operated 140,000 non-stop flights on January 1, which was a common holiday all over the world. Of which there were over 50 million possible ways to build a one-stop itinerary. While one might argue that an algorithm would not be needed if filtered combinations of transferring flights were explicitly listed out, reducing it from a path-finding problem to a path-planning problem (Sommer, 2014), the later approach was neglecting the fact that the availability of a flight was dynamic due to reasons such as flight re-scheduling, seats sold-out, and changes of price. Moreover, casting it as a path-finding problem allowed for personalization.

Certainly, the search problem would become much easier and thus not requiring a sophisticated algorithm if the number of airlines joining the interlining framework was very small, however, this was not the concern of this paper and instead, all airlines in the year 2013 dataset were assumed in.

In this work, a model was built to output some transfer airport choices given a departure airport and an arriving airport. Each choice came with a probability score such that those with higher scores were more likely to be valid choices and should be considered first at path-finding. The model was trained to be a heuristic function which captured the airports' relationship based on possible flight transfers as defined by some rules, using the year 2013 historic flight schedule mentioned.

Path-finding algorithms were covered in the first part of the literature survey, ending up with the introduction of heuristic function for reducing the complexity of the problems, followed by, in the second part, representation learning for producing richer embedding representations for objects in the graph of a path-finding problem. The idea of heuristic function and embedding learning formed the core idea of the solution. Then, the methodology was presented, including the challenge in flight interlining path-finding, the proposed solution, details about the data, and how the trained model was evaluated. Lastly, in the result section, the performance of the model was reported together with some experiments probing for possible future works.

Literature survey

Path-finding

Path-finding had been a long explored topic. In 1752, mathematician Leonhard Euler abstracted the "Seven Bridges of Königsberg" problem (Shields, 2012) as a graph. In the graph, the land masses which the bridges connected were represented as vertices (or nodes), whereas the bridges as edges. Such abstraction was a very important contribution which remained adopted nowadays. A graph of such could further be represented as a matrix, a tabular form, where the matrix element m_{ij} in the i-th row and j-th column of the table could be any numeric weight value representing the edge from vertex i to vertex j. For example, it could be just 0 or 1 for whether an edge existed between the vertices, or it could be more informative if it was traveling time, or traveling cost.

One popular path-finding problem was the "Traveling Salesman Problem" (TSP). It asked for the shortest path to visit all required vertices (or cities) exactly once and back to the origin vertex. It was a NP-hard problem, meaning that any algorithm would take, at worst, an amount of time growing polynomially fast with the number of vertices (V). Therefore, when V was large, more sophisticated algorithms, such as those incorporating heuristics (Fu, Sun, & Rilett, 2006), would be needed.

All-pairs shortest-path (APSP) and single-source shortest-path (SSSP) algorithms were the categories that most shortest path algorithms would fall into (Madkour, Aref, Rehman, Rahman, & Basalamah, 2017).

The Floyd-Warshall algorithm (Floyd, 1962), an example of APSP, computed the optimized weights m_{ij} between any two vertices i and j. The weights could either be positive or non-positive, and the algorithm recursively compared the current optimized weight m_{ij} for going from vertex i to vertex j with a new summed weight by a new path i-k-j, and if the new weight value was more optimized, it updated the m_{ij} with it. Such operation of weight optimization was called edge relaxation. Since it iterated through each i-k-j path once, it had a time complexity of $\mathcal{O}(V^3)$. However, if the graph contained negative cycles, the algorithm could computationally fail as exponentially large numbers may occur (Hougardy, 2010).

The Bellman-Ford algorithm (Bellman, 1958), which was a SSSP, accepted positive and non-positive weights. For a graph of V vertices and E edges, it calculated the optimized weights for all vertices pairs. It had a time complexity of $\mathcal{O}(EV)$ as it repeated edge relaxation process for $V-1$ times for each edge. If it was a complete graph (i.e. each vertex was connected to each of the other vertices), the time complexity became $\mathcal{O}(V^3)$.

Unlike the Bellman-Ford algorithm, the Dijkstra's algorithm (Dijkstra, 1959) made use of a priority queue to make sure all vertices were visited not more than once, reducing the time complexity to $\mathcal{O}(V + ElogV)$.

To further save time, by exploring only good vertices, the A* algorithm (Hart, Nilsson, & Raphael, 1968) added heuristic weights h_{ij}, such that the final weight from any vertex i through vertex j to the goal vertex k was changed to $m_{ij} + h_{ik}$. Prior knowledge could be considered by the algorithm through the heuristic weights, such that hopeful vertices (those along the actual best path) carried smaller heuristic weights (when smaller weights were in favor). Compared to the heuristic-free Dijkstra's algorithm, the A* was more selective in what vertex to explore, thus saving unnecessary trials. It was reported that the A* and other Dijkstra-based algorithms were the most preferred (Kumari & Lobiyal, n.d.) path-finding algorithms, and efforts had been put on creating new Dijkstra-variants (Idwan & Etaiwi, 2011; Lotfi et al., 2021).

All aforementioned algorithms served to find the shortest path, the Yen's K-shortest paths algorithm (Yen, 1970), in contrast, utilized any of those algorithms to retrieve not just the shortest one, but the K-shortest ones. It deviated from a way-vertex of the (n-1)-th shortest path, and from that vertex onward, applied a shortest path algorithm to find the n-th shortest path. This algorithm would be useful when not only the best path was needed.

Embedding representation learning for graph

While a scalar weight had been repeatedly used to represent an edge (or a node-to-node relationship), recent development in machine learning had given rise a richer, vector form of representation.

Node2vec (Grover & Leskovec, 2016) was an algorithm to construct a vector for each node. It used a technique called random work to randomly sample, for each node, a set of paths from the graph as the training data. Its advantage was that both breadth-first paths and depth-first paths were drawn for learning, respectively, the local and global information about a node. As a node was to a word as a path was to a sentence, node2vec learnt node embeddings by using the same strategy called skip-gram (Mikolov, Chen, Corrado, & Dean, 2013), which masked the node to-be-learnt and trained the neural network to reproduce it. First used in speech recognition, and being borrowed to node2vec, the technique of skip-gram was also widely used in other fields, such as protein classification (Islam, Heil, Kearney, & Baker, 2018) and biological pathway similarity search (Zhang, Kwong, Liu, Lin, & Wong, 2019), that dealt with sequential data. However, node2vec could not address the cold-start problem (Kazemi & Abhari, 2020), and it could also not deal with dynamically changing graph (Zhou et al., 2018) as it could only capture what was seen from the previous training data.

There had been a rapid development in Graph Neural Network (GNN), which already had a wide range of applications including knowledge graph, recommendation system, social network, traffic network, and so on (Asif et al., 2021). One unique feature to the learning of a GNN was owing to its mechanism to propagate information across the graph, such that sophisticated properties in edge, node, and graph level could be computed (Shlomi, Battaglia, & Vlimant, 2021) for regression and classification tasks. By their architectures, GNNs could be categorized (Wu et al., 2021) into convolutional GNN, recurrent GNN, spatial-temporal GNN, and graph autoencoder.

In particular, the autoencoder framework consisted of an encoder and a decoder, where the encoder was responsible for converting an input to an embedding, and the decoder reversed the embedding back to the input. The autoencoder was trained to minimize the loss (or difference) between the original input and the reversed one, and the smaller the loss, the better the intermediate embedding was representing the original input. Kipf and Welling (2016) proposed a variational autoencoder using a convolutional GNN as encoder and a simple dot-product as decoder, while Pan et al. (2018) adopted a generative adversarial networks (GANs) to regularize a convolutional GNN based auto-encoder in order to learn more robust node embedding representations. It was also shown (Haddad & Bouguessa, 2021) that some autoencoders were able to capture five topological features of graphs which made the method superior to other methods.

Methodology

Challenge of the problem

There were two goals for this work. First, it was to learn airport embeddings that captured the relationship between the airports, based on their possibilities to form departure-transferring-arrival (DTA) journeys. Secondly, it was to have an algorithm for suggesting some transferring airports given a pair of departure-arrival (DA) airports. The algorithm could then be used as a heuristic function for A*-like path-finding purpose.

Casting the flight interlining problem as a graphical problem, a flight became an edge, but a node was not simply an airport, but an airport at a specified time that the flight departed from or arrived at. It added a new node even for the same airport each time a new flight (a new edge) connecting the airport was added to the graph.

In real-world application, it was usually configurable for how long the waiting time between the two flight legs could be. The longer it was, the bigger the graph would grow into, making it difficult to be solved by conventional algorithms as their time complexity depended on the size of the graph.

On the other hand, the representation learning methods mentioned in the literature survey would result in the same number of embeddings as the number of nodes, which was not the number of airports, it was thus difficult if not impossible to find a single generalized representation for each airport that represented its relationship with the other airports.

Proposed solution

To tackle the problem, first, the path sampling idea in node2vec was used here to generate possible DTA journeys that complied with some rules. To simplify the problem, only the time duration of the DTAs was kept for ranking purpose, but the times themselves were dropped so an airport was not distinguishable by any time.

A simple triplet neural network was proposed for training airport embeddings from the DTAs. Owing to the nature of the triplet network, the same network could be used to convert the original embeddings of some airports (such as transferring airport candidates) to their projected embeddings. With them, a k-Nearest-Neighbor (kNN) algorithm could determine which of the candidates were more likely to be the correct transferring airports. To simplify the kNN search, only airports reachable from the departure airport were considered in the process, instead of all airports in the world.

Therefore, the first goal and half of the second would be served by the triplet network alone, and the kNN algorithm would complete the puzzle.

Triplet Network for generating embeddings

Siamese network (BROMLEY et al., 1993) and Triplet network were famous for similarity tasks. The Siamese neural network enabled image classification (Koch, 2015; Roy, Harandi, Nock, & Hartley, 2019) by similarity. It had a twin network – sharing the same set of parameters – such that any two photos of the same class passing through the two legs of the network would result in highly similar embeddings. It was used in face verification for security clearance, for instance.

The triplet network (Hoffer & Ailon, 2015), however, had three identical networks for taking an anchor (A), a positive (P), and a negative (N) input. The network was so trained to produce a high similarity score between the photos of the anchor-positive pairs (AP), and a low similarity for those in the AN pairs. Such discriminating power of true positives against false positives was why it was chosen for this work.

As illustrated in figure 1, the three identical projection networks P took as inputs $V-D$(as positive, P), $A-D$(as anchor, A), and $NV-D$(as negative, N), where D, A, V, and NV were, respectively, the embeddings of a departure airport, an arrival airport, a transferring airport, and any airport that was reachable from the departure but not connecting to arrival airport. The three inputs had all been subtracted by D, which was motivated by the intention to make A close to V but distant from any NV when D was the focus airport(figure 2).

Triplet loss

The triplet loss (Chechik, Sharma, Shalit, & Bengio, 2010) was formulated as [1].

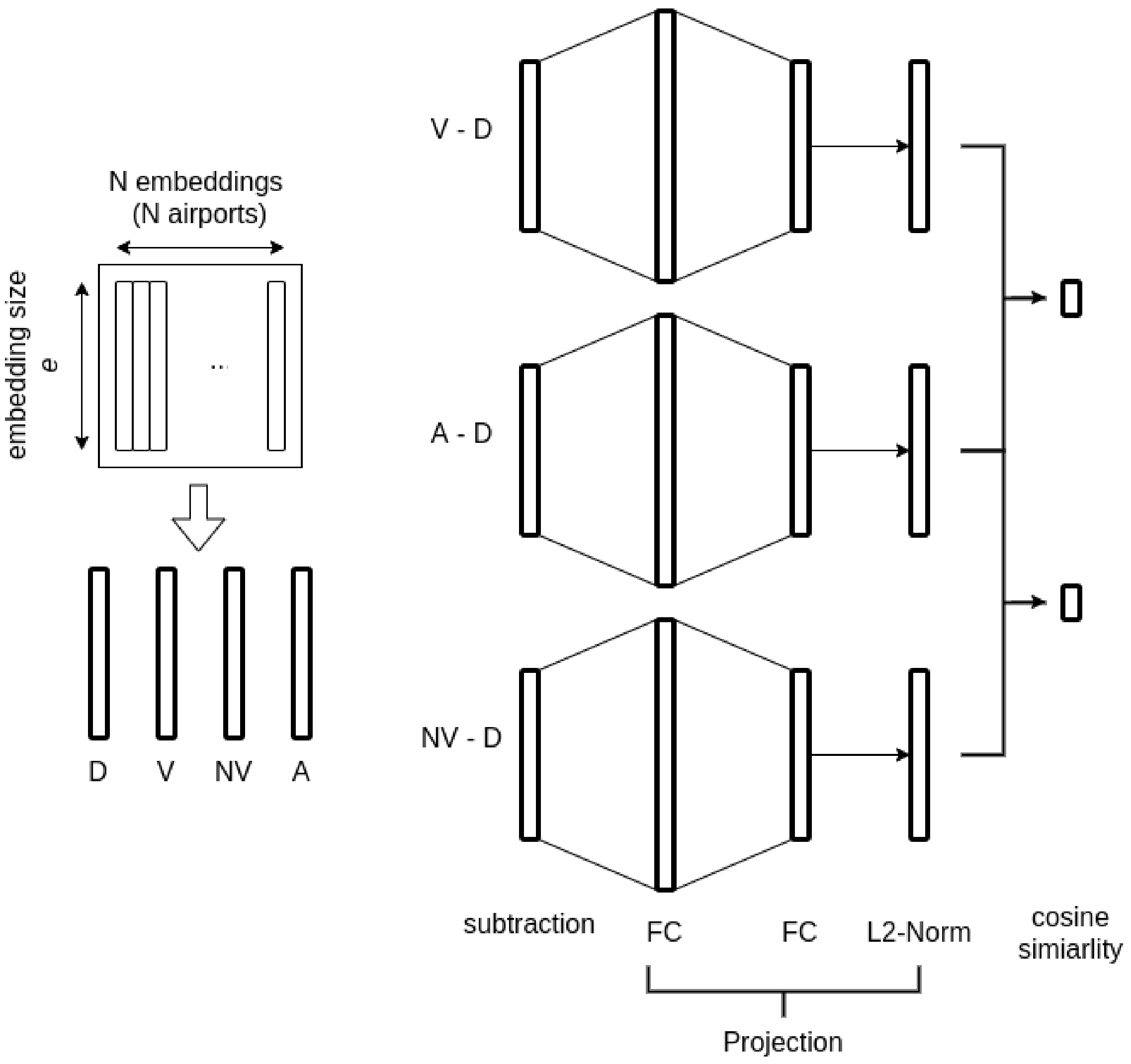
$$L = \max(0, \text{dist}(P(A-D), P(V-D)) - \text{dist}(P(A-D), P(NV-D)) + M) \tag{1}$$

where $\text{dist}(\cdot,\cdot)$ was the one minus cosine-similarity of a pair of AP or a pair of AN, which would result in $+2$ for the largest distance and 0 for the smallest. M was a margin to partially decide what APN triplets would contribute to the loss, and only the embeddings of the contributing airports would be updated in the learning process.

Figure 3 visualized the triplet loss and how it selected embeddings (airports) to update. For each pair of AP, all its Ns could be categorized into hard, semi-hard, and

Figure 1

Model architecture



easy negatives according to their distances to the A. When a N had a smaller distance to the A than the P, it was a hard negative because a negative was supposed to be farther away from the A than the P was from the A. When a N was within the marginal area, it was called semi-hard, and depending on the size of the margin, an N residing outside of the margin would become an easy negative. Easy negatives were not contributing to the loss as its loss was smaller than 0 and thus omitted by the max(·) function in [1].

The ability to omit some airports allowed the training process to focus on the airports breaking the APN relations, and moved them around such that the positive (correct transferring airport) and the negative were apart from each other by a margin, and so came the discriminating power of the triplet network.

The hard negatives were distinguished from the semi-hard, because learning the hard negatives, according to Schroff, Kalenichenko, and Philbin (2015), could lead the algorithm to some bad local minima in the early stage of training. To prevent this, semi-hard negatives were learnt on by the model in the early stage of training, and hard negatives were used later.

Online triplet loss. There were two ways to find hard or semi-hard negatives, offline or online. The offline approach required computing the distances among all samples, which was expensive ($\mathcal{O}(N^2)$ when the sample size was large. With over a million samples for this work, the online approach was used. Since samples were sent to training in batches, the online approach only calculated the distances among a batch of samples which was much smaller and therefore efficient, however, it did not guarantee important semi-negatives be found and so it was also less effective.

Weighted triplet loss. Triplet loss had good discriminating power, but it did not differentiate among the positives. The goal of this paper was to build a model that suggested true transferring airport, not ranking them, but if the rank was important, then that differentiation power was crucial. In light of this possible demand, a modified version of triplet loss was created on top of an existing python code implementation of the original triplet loss. It allowed different weight values for different AP pairs according to the Ps' importance. Effectively, it modified [1] into [2].

$$L = \max(0, w \cdot \text{dist}(P(A-D), P(V-D)) - \text{dist}(P(A-D), P(NV-D)) + M) \quad (2)$$

, where $w = w(A, V) = \frac{\pi}{4} \cdot \text{Rank}(V|A, D)$ was the weight factor, and $\text{Rank}(V|A, D) \in [0, 1)$ was a ranking value to be defined in the data-preprocessing section.

k nearest neighbor

Given the embeddings of a departure airport D, an arrival airport A, and a list of airports VS reachable from the departure airport, the model could produce a list of recommended transferring airports, sorted by the cosine-similarity function $S(\cdot, \cdot)$ as [3].

$$\underset{V \in VS}{\operatorname{argsort}} S(P(A-D), P(V-D)) \tag{3}$$

Data

The historic flight schedule records in year 2013 (OAG Aviation Worldwide LLC, 2015) consisted of almost 9 million flights ran by 874 carriers in over 4,000 airports around the world. Each record was described by at least a flight number, the departure and arriving airports, the number of stopping airports, the local departure and arriving dates and times, the period in which the flight was available, and the day of week that the flight was available in the period.

Data pre-processing

Only flights with zero stop (i.e. direct flight) and operated in the first quarter of the year were used. Since each record represented a group of flights during its period of availability, the records were expanded to become a list of individual flight of its own departure and arrival airports and times.

Then, DTAs were extracted such that the departure time at the transferring airport was 30 minutes to 1 day after the arrival time. Besides the names of the airports, the journey duration was saved for each DTA for the future steps.

Even with data only in the first quarter, 649,246,631 DTAs were discovered. Therefore, merge-sort was used to produce the list of DTAs sorted by the departure airport, arrival airport and duration. Then, it was used to group the DTAs of the same departure, arrival and transferring airports in order to get the median duration time as a measure for ranking the transferring airport V among all transferring airports VS for the same DA journey. The rank value was calculated by [4].

$$\operatorname{Rank}(V|A, D) = \frac{\operatorname{order}(V; A, D)}{|VS|} \tag{4}$$

The order$(\cdot)$ function returned a 0-based order value of the transfer airport in the ascending order of median duration, and $|VS|$ denoted the total number of transferring airports.

Data pipeline for training

All 1,151,856 grouped DTAs were used as training data because the goal was to build a heuristic function instead of a predictive function. However, the predictability was also examined through a separated experiment that divided the dataset into two, and would be covered in the results section.

At each time, the pipeline took all DTAs of the same DA, and produced $s > 2$ training tuples for each transferring airport. A tuple was in the form of (x_0, x_1, y), where x_0 was always the departure airport. The x_1 for the first and the second tuples were always the arrival airport and the transferring airport, whereas the rest of the x_1's were chosen from the following two sources. First, airports reachable by the departure airport were picked randomly, and if there was still a vacancy after all the aforementioned airports were exhausted, then all the rest of the airports were chosen randomly. y was the label and it had the same label value for the first two tuples, and distinct values for the rest.

The final size of a training batch was an integral multiplication of s.

Model Implementation and the training path

The model was constructed using the tensorflow package and the tensorflow_addon package. The 2 FC layers in the project network P had 512 and 256 nodes respectively and they were without bias. The size of each embedding was $e = 256$ and they were initialized randomly. The same margin M of 0.3 was used for both the online semi-hard, and hard triplet loss. The Adam optimizer (Kingma & Ba, 2014) was selected for this work as it used inertial methods against any rapid changes (Barakat & Bianchi, 2021). The Adam optimizer's learning rate was set to be 0.05, and the β_1 and β_2 were set as 0.9 and 0.999 respectively. In the data pipeline, $s = 4$ was set to use two false transferring airports for each true transferring airport, and 128 DTAs were included in each batch such that the batch size was 512.

The model was first trained using the online semi-hard triplet loss for 20 epochs, followed by another 20 epochs using the online hard triplet loss. Several experiments were done by varying the training path, embedding size e, and the triplet loss margin

M for performance comparison.

Evaluation methods

Given that learning a heuristic model rather than a predictive one was the target of this paper, the model was both trained and evaluated on the whole dataset.

Recall

Since the objective was to suggest true transferring airports, and not the ranking among the suggestions, the averaged Recall@$(k + tol)$ ([5], ranged from 0 to 1) was adopted as the primary evaluation metric to account for the relevance of the top-$(k + tol)$ suggestions.

$$\text{Recall@}(k + tol) = \frac{\#(\text{Transferring Airports In Top-}(k + tol)\ \text{ Suggestions})}{k} \tag{5}$$

k was the total number of true transferring airports, and tol was the tolerance value for expanding to include tol more top suggestions for evaluating the recall value. The higher the tolerance value, the higher the chance that more true transferring airports originally left out from the top-k would be included back. [5] was evaluated for each DA pair and the averaged value of them was used as the final recall score.

nDCG

To examine the quality of rankings of the suggestions, the averaged nDCG@10 ([6], ranged from 0 to 1) was chosen to be the secondary metric.

$$\text{nDCG@10} = \frac{\text{DCG@10}}{\text{iDCG@10}} \tag{6}$$

$\text{DCG@10} = \sum_{i=0}^{10} \frac{\text{rel}(i)}{\log_2 (i+1)}$ was the gain before normalized, $rel(i)$ denoted the relevance score of the i-th airport in the sorted suggestions including only the true transferring airports, and iDCG was the ideal DCG score for the perfect ordering. The relevance scores were assigned to the airports such that the true top had a score of 10, and 9 for the next, and so on. The airports after the 10-th place all had zero scores.

Baseline evaluation score

A baseline that was independent of the modeling method was established as the recall and nDCG scores for random suggestions.

For recall, the baseline for each DTA was given by [7], where N was the number of airports reachable from the departure airport.

$$\text{Recall@}(k + tol) = \sum_{i=1}^{k} i \cdot \frac{C_i^{k+tol} \cdot C_{n-i}^{N-k-tol}}{C_k^n} \tag{7}$$

The resulting baselines were listed in table 1. The baselines increased with the tolerance value because more true transferring airports could be counted towards the score. In this paper, unless specified otherwise, the recall score was always referred to the case that $tol = 0$.

Table 1

Baseline scores for Recall@$(k + tol)$

tol	Recall@$(k + tol)$
0	0.209
5	0.472
10	0.578
20	0.699

For nDCG, the baseline for each DTA was given by [8], where $k' = \min(10, k)$.

$$\text{nDCG@10} = \frac{\frac{1}{k'} \cdot \sum_{i=1}^{k'} \text{rel}(i) \cdot \sum_{i=1}^{k'} \frac{1}{\log_2(i+1)}}{\sum_{i=1}^{k'} \frac{\text{rel}(i)}{\log_2(i+1)}} \tag{8}$$

The baselines were listed in table 2 according to the number of transferring airports k of a DA pair. It was so grouped because smaller pairs tended to have higher scores even at baseline, and since they were the majority, it made the efforts for improving the ordering of suggestions less visible.

Bigger pairs were challenging especially when the training batch size was limited by the computation power, however, they could be cut down to smaller pairs by removing some possible, yet not valuable, transferring airports.

Therefore, the group where k was in the range of 2 to 5 was the best candidate for monitoring the ordering quality, and unless specified otherwise, nDCG score of that range was referred to as the nDCG score in this paper.

Table 2

Baseline scores for nDCG@10

Number of true transferring airports, k	Number of departure and arrival airports pairs	nDCG@10
2-5	135,305	0.977
6-10	25,366	0.895
11-50	15,741	0.515
51-100	598	0.131

Results and discussions

The model achieved a recall rate of 0.903 and a nDCG score of 0.921, comparing to the baseline of 0.203 and 0.895 respectively. If a tolerance of $tol = 5$ airports was allowed, the recall rate could be increased to 0.988 which was almost certain that the correct airports were among the top suggestions. All other scores were listed in table 3 and table 4 .

It was expected that the improvement for the nDCG score relative to the baseline was not significant, because the model was not powerful in differentiating and ordering the positives (the transferring airports). However, an attempt to improve nDCG by using the weighted triplet loss, as well as experiments for other purposes, would be presented next.

Table 3

Results for recall

tol	Baseline	**Result**
0	0.209	**0.903**
5	0.472	**0.988**
10	0.578	**0.995**
20	0.699	**0.999**

Experiment 1: Hyper parameters search for triplet loss margin and embedding size

6 different margins were tried, and generally their recalls increased as smaller margins were used (figure 4), likely because a small margin could focus the algorithm on more challenging negatives (false transferring airports). The nDCG scores were peaked at around a margin of 0.5 but the superiority was not significant. The final margin value of 0.3 was chosen in this paper as a balance of the two.

For embedding size, the recalls were always increasing with it, because a larger size would give the model more degrees of freedom to learn from the data. However, a large size would increase the training time and memory usage, thus 256 was selected as it had gained the most recall improvement with the least increase in embedding size.

Experiment 2: Training path search

3 paths were tried, following the practice that the semi-hard triplet loss was used in the early stage, before the hard triplet loss could be used to finalize the training. The paths were different in terms of the number $s-2$ of false transferring airports used per DTA.

All paths started with phase 1. Path 1 then ended in phase 1.1. Path 2 moved on to phase 2 before finishing with phase 2.1. Path 3 moved on to phase 2, 3 and 3.1 in order. The definitions for the phases were listed in table 5.

Table 4

Results for nDCG

Number of true transferring airports, k	Number of departure and arrival airports pairs	Baseline	**Result**
2-5	135,305	0.977	**0.983**
6-10	25,366	0.895	**0.921**
11-50	15,741	0.515	**0.605**
51-100	598	0.131	**0.287**

Generally speaking, as shown in figure 5, gradual increasing of s while staying with semi-hard triplet loss could continuously improve both the recall and nDCG scores, partially because increasing s had made more false transferring airports available to the training process to constraint the airport embeddings more effectively.

While the hard triplet loss always boosted the recall as it got flattened out, it also always decreased the nDCG score. On the other hand, during the hard triplet loss training stage, the recall scores tended to increase in the same way with the number of training epochs, regardless of the path.

Experiment 3: Improving nDCG with weighted triplet loss

The upper right plot in the figure 6 showed that there was a significant improvement for nDCG in terms of both its value, and its lasting increasing trend. While it did not deteriorate the performance on recall, it strongly suggested that the weighted triplet loss could replace the non-weighted version for training a model with good ordering quality. However, as the bottom plots showed, such improvement became less observable for the bigger groups, which could be partially accounted for by the small training batch size comparing to the group size.

Table 5

Phases of training paths

phase	s	triplet loss
1	4	semi-hard
1.1	4	hard
2	8	semi-hard
2.1	8	hard
3	16	semi-hard
3.1	16	hard

Experiment 4: the model's predictability

To examine the potential of making predictions with the model, 10% of all training data became a test dataset and was kept away from the training process. The model was then trained with the rest of the data, and evaluated on the training data and the test data separately.

Figure 7 showed that there was a 0.635 recall rate on the test set, indicating some degree of predictability from the model. While such predictability depended on the architecture of the model, it could also be affected by the noise in the data, and it reflected to what degree the DTAs discovered in the data pre-processing was following a consistent underlying pattern.

Conclusion and future works

This work demonstrated a heuristic function for flight interlining path-finding, achieving an averaged recall of 90.3% without any tolerance. With a tolerance of 5 airports, however, it could even be increased to 98.8%, meeting with the goal of suggesting the correct transferring airports out of all possible ones. It was also an evidence that the underlying airport relations based on the DTA generation rules could be captured by a model.

The ordering of the suggestions was also examined using the nDCG score. It was not surprising that the improvement in nDCG relative to the baseline was low when it was trained with the traditional triplet losses, because they did not differentiate between the positives.

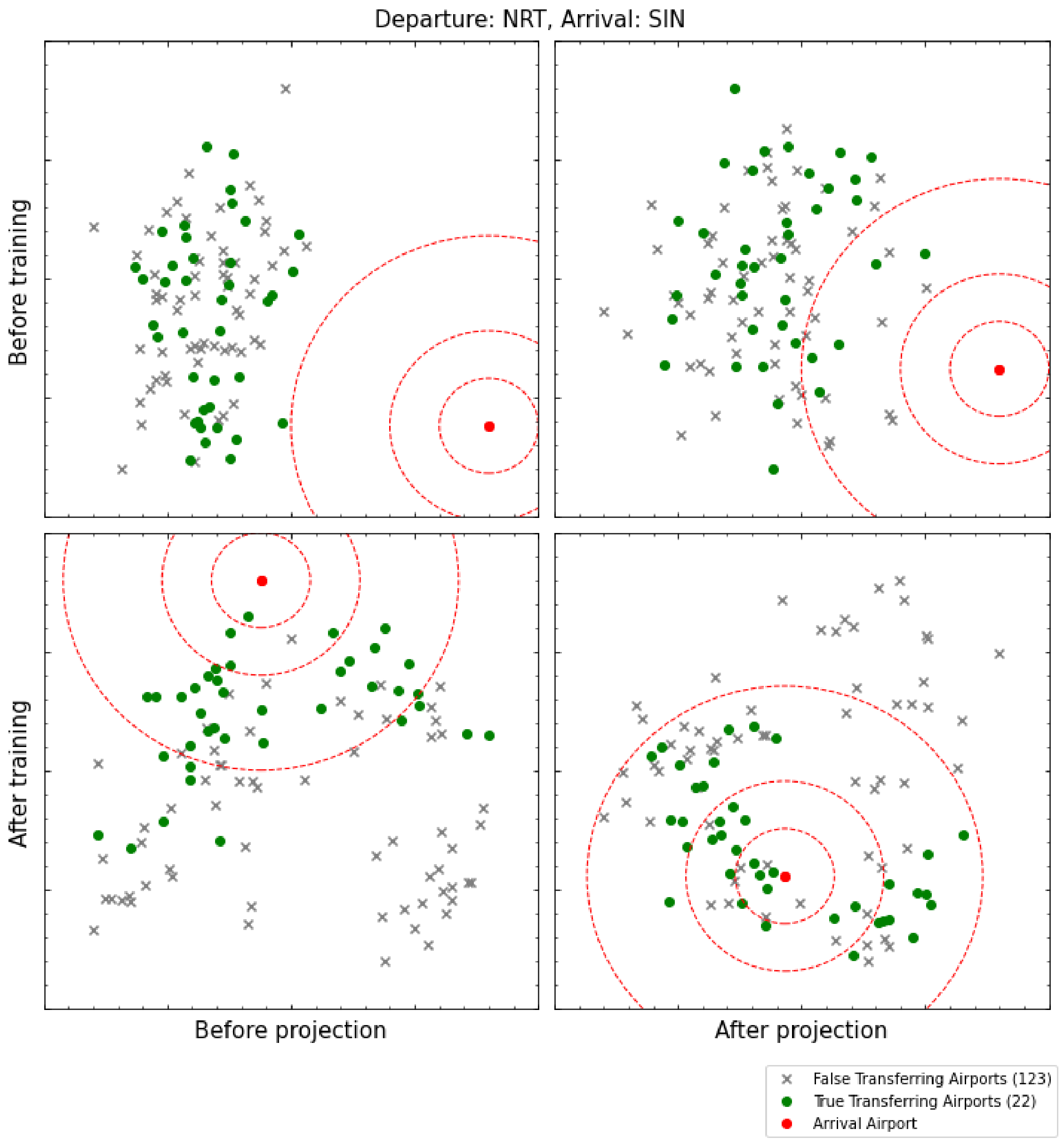
Therefore, a weighted version of the triplet losses was created, on top of an existing python code implementation, to reflect the difference among the transferring airports. Figure 6 strongly suggested that the weighted triplet losses outperformed the original one in terms of establishing ordering within the positives, and was worth further exploration to make the heuristic function not just giving out correct suggestions but also better suggestions first.

Figure 7 revealed that the model carried some degree of predictability for the dataset so constructed. While it was not the intention for the model to be predictive, altering of the architecture and further polishing of the dataset were possible to increase the prediction accuracy for a wider application of the model.

Lastly, the transferring airports in this work were generated from a history flight schedule following a few rules, so the model was capturing about the rules and how the flights were scheduled. However, if the data came from history customers' transferring records, or the web browsing history when customers were looking for transfer flights, then such user-centric data could certainly fuel a model making personalized suggestions.

Figure 2

Distribution of transferring and non-transferring airports in reduced dimensions



Note. An example showing a case of choosing Tokyo's NRT as the departure airport and Singapore's SIN as the arrival airport. First column: transfer airports (green dots) became (i) closer to the arrival airport (red dot) and (ii) more separated from the other airports (gray crosses) after the model had learnt from the dataset. Second row: the projection network P served to move the transfer airports closer to the arrival airport.

Figure 3

Illustration of hard, semi-hard, and easy negatives

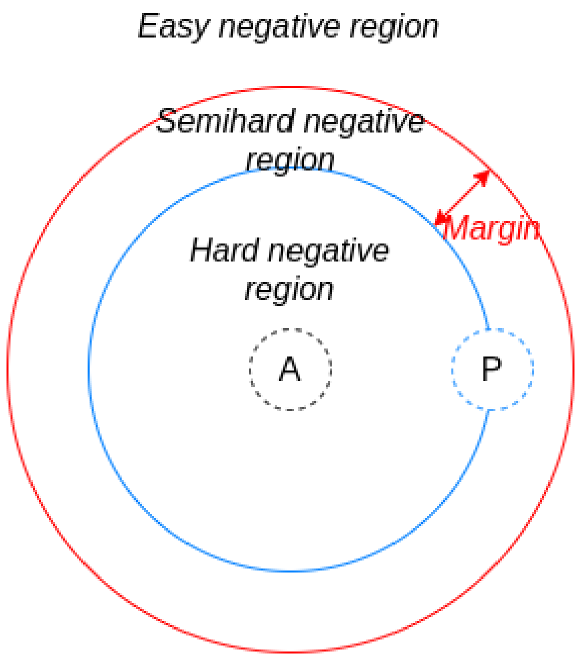
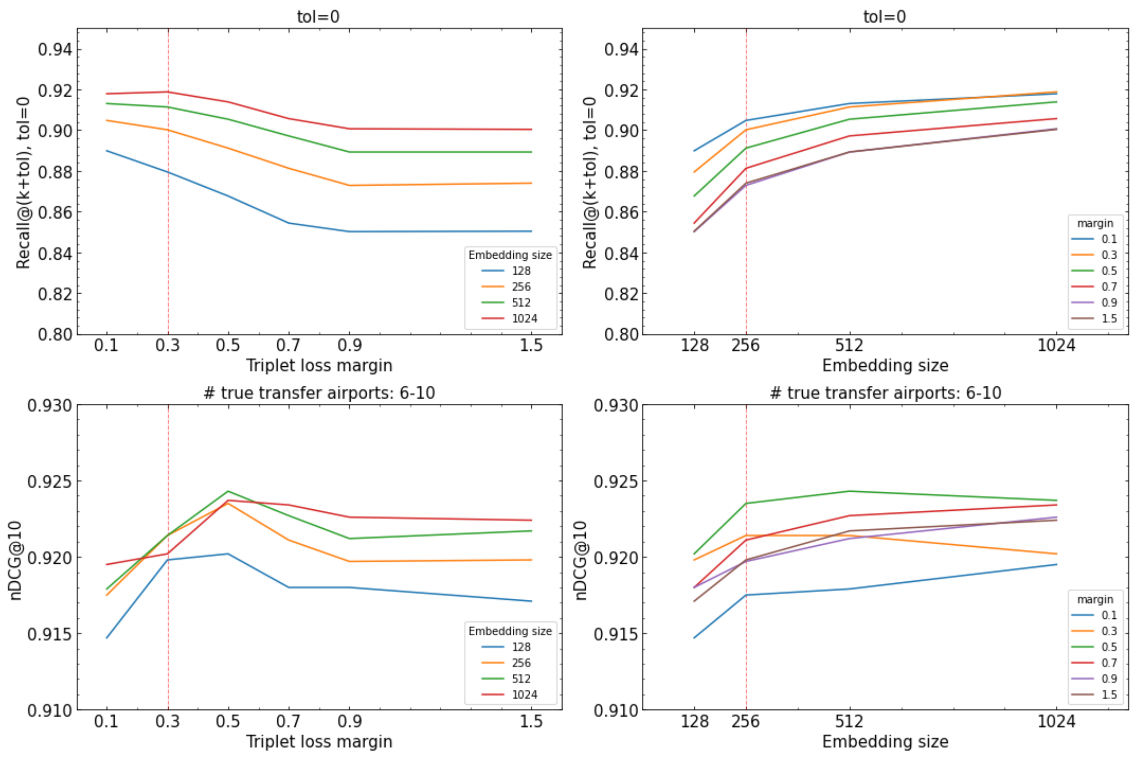


Figure 4

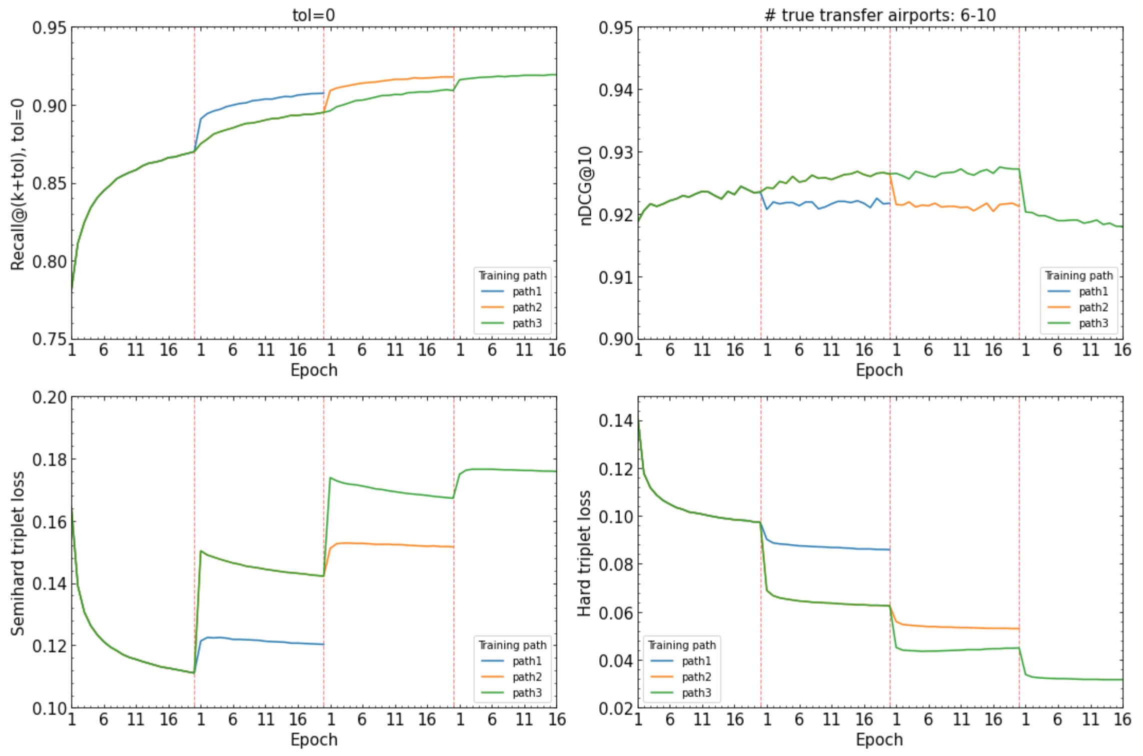
Comparison of different configurations of triplet loss margin and embedding size



Note. The top plots showed how the recall changed with the margin and the embedding size. The bottom plots included the change for nDCG.

Figure 5

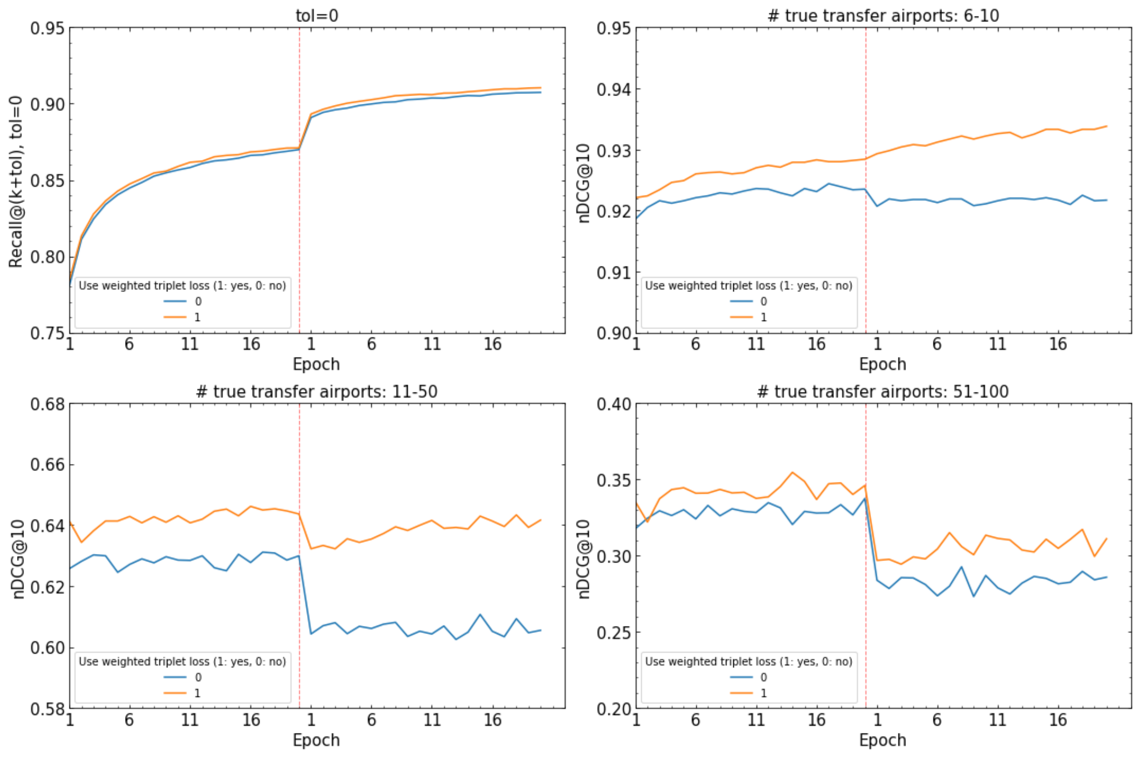
Comparison of different training paths



Note. The top plots were for the recall and nDCG scores. The bottom plots were for the semi-hard and hard triplet loss curves. The red dashed line divided the plots into phases. The green line in the first three blocks were for phase 1, 2, and 3. The blue line in the second block was for phase 1.1, whereas the orange line in the third block was for phase 2.1. Finally, the green line in the last block was for phase 3.1.

Figure 6

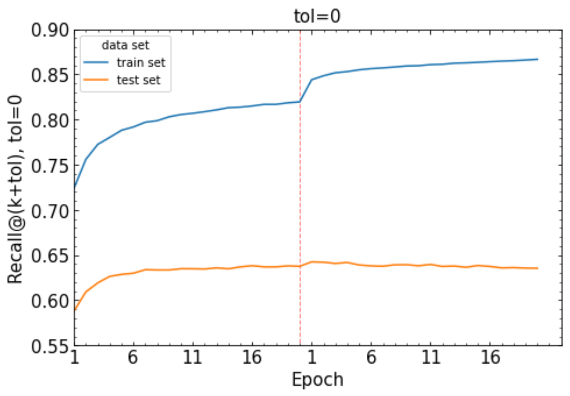
Comparison of recall and nDCG with and without weighted triplet loss



Note. The top plots were for the recall and nDCG scores. The bottom plots were for the semi-hard and hard triplet loss curves. The red dashed line divided the plots into phases. The green line in the first three blocks were for phase 1, 2, and 3. The blue line in the second block was for phase 1.1, whereas the orange line in the third block was for phase 2.1. Finally, the green line in the last block was for phase 3.1.

Figure 7

Predictability



References

Asif, N. A., Sarker, Y., Chakrabortty, R. K., Ryan, M. J., Ahamed, M. H., Saha, D. K., … Tasneem, Z. (2021). Graph neural network: A comprehensive review on non-euclidean space. *IEEE access*, *9*, 60588–60606.

Barakat, A., & Bianchi, P. (2021). Convergence and dynamical behavior of the adam algorithm for nonconvex stochastic optimization. *SIAM journal on optimization*, *31*(1), 244–274.

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, *16*(1), 87-90.

BROMLEY, J., BENTZ, J. W., BOTTOU, L., GUYON, I., LECUN, Y., MOORE, C., … SHAH, R. (1993). Signature verification using a siamese' time delay neural network: Advances in pattern recognition systems using neural network technologies. *International journal of pattern recognition and artificial intelligence*, *7*(4), 669–688.

Chechik, G., Sharma, V., Shalit, U., & Bengio, S. (2010). Large scale online learning of image similarity through ranking. *Journal of Machine Learning Research*, *11*(36), 1109-1135. Retrieved from `http://jmlr.org/papers/v11/chechik10a.html`

Coles, H. (2019). *The future of interline: A new model for seamless journeys.* Retrieved from `https://www.iata.org/contentassets/23426d4b09a0446dbe831601869098a1/future-of-interline-wp.pdf`

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271.

Floyd, R. (1962). Algorithm 97: Shortest path. *Communications of the ACM*, *5*(6), 345.

Fu, L., Sun, D., & Rilett, L. (2006). Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & operations research*, *33*(11), 3324–3343.

Grover, A., & Leskovec, J. (2016). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd acm sigkdd international conference on knowledge* *discovery and data mining* (pp. 855–864). ACM.

Haddad, M., & Bouguessa, M. (2021). Exploring the representational power of graph autoencoder. *Neurocomputing (Amsterdam)*, *457*, 225–241.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on systems science and cybernetics*, *4*(2), 100–107.

Hoffer, E., & Ailon, N. (2015). Deep metric learning using triplet network. In *Similarity-based pattern recognition* (pp. 84–92). Cham: Springer International Publishing.

Hougardy, S. (2010). The floyd–warshall algorithm on graphs with negative cycles. *Information processing letters*, *110*(8), 279–281.

Idwan, S., & Etaiwi, W. (2011). Dijkstra algorithm heuristic approach for large graph. *Journal of applied sciences (Asian Network for Scientific Information)*, *11*(12), 2255–2259.

Islam, S. M. A., Heil, B. J., Kearney, C. M., & Baker, E. J. (2018). Protein classification using modified n-grams and skip-grams. *Bioinformatics*, *34*(9), 1481–1487.

Kazemi, B., & Abhari, A. (2020). Content-based node2vec for representation of papers in the scientific literature. *Data & Knowledge Engineering*, *127*, 101794. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0169023X1830185X` doi: https://doi.org/10.1016/j.datak.2020.101794

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization.

Kipf, T. N., & Welling, M. (2016). Variational graph auto-encoders.

Koch, G. R. (2015). Siamese neural networks for one-shot image recognition..

Kumari, A., & Lobiyal, D. (n.d.). Efficient estimation of hindi wsd with distributed word representation in vector space. *Journal of King Saud University. Computer and information sciences.*

Lotfi, M., Osorio, G. J., Javadi, M. S., Ashraf, A., Mahmoud, M., Samih, G., & Catalao, J. P. S. P. S. (2021). A dijkstra-inspired graph algorithm for fully autonomous tasking in industrial applications. *IEEE transactions on industry applications*, 1–1.

Madkour, A., Aref, W. G., Rehman, F. U., Rahman, M. A., & Basalamah, S. (2017). A survey of shortest-path algorithms.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space.

OAG Aviation Worldwide LLC. (2015). Oag2013.zip. In *Oag yearly historic flight schedules.* Harvard Dataverse. Retrieved from `https://doi.org/10.7910/DVN/COHFWA/NWNCX0` doi: 10.7910/DVN/COHFWA/NWNCX0

Pan, S., Hu, R., Long, G., Jiang, J., Yao, L., & Zhang, C. (2018). Adversarially regularized graph autoencoder for graph embedding.

Roy, S., Harandi, M., Nock, R., & Hartley, R. (2019). Siamese networks: The tale of two manifolds. In *2019 ieee/cvf international conference on computer vision (iccv)* (pp. 3046–3055). IEEE.

Schroff, F., Kalenichenko, D., & Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering..

Shields, R. (2012). Cultural topology: The seven bridges of königsburg, 1736. *Theory, culture & society*, *29*(4-5), 43–57.

Shlomi, J., Battaglia, P., & Vlimant, J.-R. (2021). Graph neural networks in particle physics. *Machine Learning : Science and Technology*, *2*(2).

Sommer, C. (2014). Shortest-path queries in static networks. *ACM computing surveys*, *46*(4), 1–31.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE transaction on neural networks and learning systems*, *32*(1), 4–24.

Yen, J. Y. (1970). An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of applied mathematics*, *27*(4), 526–530.

Zhang, J., Kwong, S., Liu, G., Lin, Q., & Wong, K.-C. (2019). Pathemb: Random walk based document embedding for global pathway similarity search. *IEEE Journal of Biomedical and Health Informatics*, *23*(3), 1329-1335. doi: 10.1109/JBHI.2018.2830806

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., . . . Sun, M. (2018). Graph neural networks: A review of methods and applications.