Flight Radius Algorithms

Assia Kamal Idrissi, Arnaud Malapert, Rémi Jolin

▶ **To cite this version:**

Assia Kamal Idrissi, Arnaud Malapert, Rémi Jolin. Flight Radius Algorithms. 8th International Conferance on Operations Research and Entreprise Systems, Feb 2019, Prague, Czech Republic. hal-02069634

HAL Id: hal-02069634
https://hal.science/hal-02069634

Submitted on 15 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Flight Radius Algorithms

Assia Kamal Idrissi[1,2], Arnaud Malapert[2] and Rémi Jolin[1]
[1]*Milanamos, 1047 route des Dolines, Sophia Antipolis, France*
[2]*Université Côte d'Azur, CNRS, I3S, France*

Keywords: Flight Radius Problem, Time-independent Model, Graph Database, Shortest Path Algorithms.

Abstract: In this article, we present the flight radius problem (FRP) on the condensed flight network (CFN). Then, giving a specific flight that is defined by an origin and destination (OD) pair, the problem consists in finding routes that connect the OD pair and satisfy a regret constraint on time, distance or cost. The found routes help airline manager to find business opportunities. This problem arises in the real world, for instance in some air transportation companies. The FRP is formulated as finding a maximal subgraph of nodes belonging to routes satisfying a regret constraint. Such routes can be found using shortest paths algorithms (SPA). The CFN is generated using a time-independent approach and stored in the graph database `Neo4j`. Implementing SPA in `Neo4j` is challenging since the graph database stores the weights of the graph in a separate data structure. In this paper, we propose four methods to solve the FRP: these methods combine parallel and sequential processing with more optimization to overcome time and memory costs. The experimental evaluation demonstrates that the best algorithm is the extended Dijkstra algorithm which meets the real-time constraints of the targeted industrial application.

1 INTRODUCTION

The growth of air passenger needs has forced airlines to improve their quality of service. The airlines should offer flights that match with preferences of passengers. For this reason, most airlines are interested in the quality of service models called QSI. This is a market share model used by airlines to estimate their part of the market. The model determines the probability a traveler selects a specific itinerary connecting an airport pair based on a list of criteria (Jacobs et al., 2012). Then, let us suppose an airline network, where nodes are the airports and arcs represent flights. Each arc is associated with a time, distance, and cost. The question is to decide if it is interesting to add a flight between a pair of airports (OD) in that network. More precisely, adding a new arc would allow passengers to make itineraries which are a little bit longer but cheaper than going directly to their destination? For this, the flight radius problem (FRP) consists in finding routes that pass by a potential arc and satisfy a regret constraint. This constraint aims to model passengers preferences. Practically, there are many criteria to be considered but the three considered in this study are time, distance, and cost. The FRP is derived from `PlanetOptim` application developed by the company *Milanamos* which is a startup specialized in air transportation. This application is a decision tool for airline managers to analyze and simulate a new market using QSI models. Then, given a specific flight that is defined by an OD pair, the process in the application starts by finding relevant airports with respect to the specific flight and then estimates market share for each route connecting the origin to the destination. Following this, the airline manager makes a decision about adding the new flight.

Motivating Example. Let us consider this example for which a new route will be created between `NCE` and `BKK` airports (see Figure 1). This new route allows passengers coming forward `NCE` to reach other airports via `BKK`. Taking the new flight can be a little bit longer but cheaper, than going directly to the final destination. The choice depends on the passengers since they have different preferences. Actually, the application returns some uninteresting airports. For instance, going from `CDG` to `DXB` via `BKK` is not a realistic itinerary for a passenger. The itinerary is too long and expensive than going directly. Then, the idea of the FRP is to filter the routes that do not match over preferences. For this reason, these preferences should be integrated into this problem.

Figure 1: Screen-shot of `PlanetOptim` application. CDG: Paris, DXB:Dubai, NCE:Nice, BKK:Bangkok, dashed airports represent uninteresting airports.

The FRP is formulated as finding a maximal subgraph such that for each node there exists a path that satisfies the regret constraint. The regret is defined for each criterion. Such paths can be retrieved using shortest paths algorithms. We simply begin by working on the CFN, which is stored in a graph database `Neo4j`. This database uses a graph structure and its graphical interface visualizes easily the airline network. However, `Neo4j` stores each node's and arc's properties separately: accessing those properties is relatively slow. Then, implementing the SPA in a graph database like `Neo4j` is challenging.

In this paper, we propose four methods to solve the FRP on the CFN. The first two are algorithms that decompose the FRP in shortest path problems and solve them in parallel using as shortest path algorithm: Dijkstra and Bellman. The third one extends the Dijkstra algorithm. However, the last one uses a Bellman algorithm that computes at once all shortest paths from both origin and destination for all criteria. We use parallel processing and sequential with more optimization to overcome time and memory costs.

The paper is organized as follows. Section 2 introduces some definitions of graph theory and some shortest path algorithms. In Section 3, we describe the CFN modelling and the graph database structure. Section 4 gives the formulation of the FRP and its properties. Section 5 describes the four algorithms. Section 6 is dedicated to experiments.

2 PRELIMINARIES

This section gives definitions of graph theory (Ahuja et al., 1993). Then, it states some shortest path algorithms.

2.1 Graph Theory

A *graph* G is a couple $G=(V,E)$ consisting of a finite set V of nodes and a set $E \subseteq V \times V$ of arcs which are ordered pairs (u,v) if the graph is directed. Each arc $(u,v) \in E$ has an associated non-negative weight $w(u,v)$. We define $|V|=n$, the order of the graph as the number of nodes meanwhile $|E|=m$ its size.

A graph $G'=(V',E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *path* is a sequence of nodes $\{v_1,v_2,...,v_k\}$ such that for each $1 \leq i < k, (v_i,v_{i+1}) \in E$ holds. If additionally $v_1 = v_k$, then the path is a *cycle*. The length of a path P is the sum of its arc weights along the path and is denoted by:

$$l(P) := \sum_{i=1}^{k-1} w(v_i,v_{i+1}).$$

We define $l^\star(s,t)$ for a given pair of nodes s and t, the length of the shortest path starting at s and ending at t. A graph G is *connected* if there exists a path joining any two nodes. A transportation network should be a connected graph.

2.2 Shortest Path Algorithms

There are two categories of shortest path algorithms: setting algorithms and correcting algorithms. The two types of algorithms are based on the labeling method and differ in the strategy of selecting labeled nodes to be scanned (Cherkassky et al., 1996).

Labeling Method. The labeled method is defined as follows. For each node u, the method maintains a distance label $d(u)$, which is an upper bound on the shortest path length to the node u, parents $p(u)$, and status $S(u)$. We have three status: *unreached*, *labeled*, and *scanned*. Initially for each node u, $d(u)=inf$, $p(u)=nil$, and $S(u)=unreached$. For a start node, the method sets $d(s)=0$ and $S(s)=labeled$. Then, the method starts by scanning labeled nodes until such node does not exist here. The `SCAN` operation of a labeled node consists in checking for all outgoing arcs $(u,v) \in E$, if $d(u)+w(u,v) < d(v)$ (see Function 1). Then, if it is, $d(v)$ is updated, $S(v)$ become *labeled*, and $S(u)$ is *scanned*. If there are no negative cycles, the arcs $(p(u),u)$ form a tree rooted at s. When the algorithm stops, the tree rooted at s is the shortest path tree.

Function 1: SCAN(u).

```
foreach (u,v) ∈ E do
    if d(u) + w(u,v) < d(v) then
        d(v) ← d(u) + w(u,v);
        S(v) ← labeled;
        p(v) ← u;
    end
end
S(u) ← scanned;
```

Setting Algorithms. The `Dijkstra` algorithm (Dijkstra, 1959) is the most known setting algorithm that works with positive weight arcs. In this algorithm, the principle is to select a node with the minimum weight at each iteration. It scans each node at most once. That leads to a complexity of $O(n^2)$ as time bound in the worst case (Ahuja et al., 1993) where n is the number of nodes. Two sets are maintained: a permanently set that represents selected nodes and a temporary set that designates nodes not yet selected. The algorithm performs the node selections operation n times. Each operation requires that it scans each temporarily labeled node which leads to $O(n^2)$. Besides, the algorithm performs the distance updates operation for all outgoing arcs of a node v. Overall, the algorithm requires $O(m)$ since each operation requires, a constant time, $O(1)$. In the end, `Dijkstra's algorithm` solves the shortest path problem in $O(n^2)$.

There are many versions of `Dijkstra's algorithm` with the aim of improving this time bound by trying different data structures and several implementations of the algorithm. In some applications of the shortest path problem, we want uniquely to determine the shortest path between two nodes. `Bidirectional Dijkstra's algorithm` solves the problem of finding the shortest path between two nodes faster since it eliminates some unnecessary computations by reducing the number of visited vertices in practice. Besides, the `A* search` is an acceleration of `Dijkstra's algorithm` in the sense that it preferably settles nodes that are closer to the destination when finding the shortest path between two nodes.

Correcting Algorithms. The `Bellman` algorithm achieves the best currently known bound of time with negative weight arcs $O(nm)$. The algorithm maintains the set of labeled nodes in a FIFO queue and allows detecting a negative cycle in a weighted directed graph. In `Bellman`, the next node to be scanned is removed from the head of the queue; a node that becomes labeled is added to the tail of the queue. The algorithm performs at most $n-1$ passes through arcs. Since each pass requires $O(1)$ computations for each arc, this implies $O(nm)$ time bound for the algorithm. `Bellman` is qualified as a robust algorithm since there is no priority queue. Some heuristics have been introduced to improve the practical performance of the algorithm. For instance, (Cherkassky et al., 1996) introduce a `parent checking heuristic` that scans a node only if its parent is not in a decrease.

3 CONDENSED FLIGHT NETWORK

3.1 Modelling

The condensed flight network (CFN) is generated from the flight timetable using the time-independent model. Nodes represent airports meanwhile the presence of an arc indicates that there exists at least one elementary connection between two airports. Time, distance, and cost labels are associated with each arc of the CFN. In our study, we omit time scheduling and keep only the transfer time represented by an arc in the graph. This technique is often used to model the information about the transfer since it is important in computing shortest paths (Delling et al., 2009).

The model consists in creating two nodes for each airport node: one to model flight departures and an other to represent flight arrivals. Then, we introduce three different types of arcs. `board_at` is inserted from an airport to departure node, `alight_at` is inserted from an arrival node to the airport, and finally a `connect_to` to model the transfer time between an arrival node and departure node of the same airport with a transfer time. In Figure 2, the graph contains four airports: `NCE`, `BKK`, `PEK`, `ICN` and four flight arcs referenced by `year_month`. Nodes in thin style represent departures, dashed nodes for arrival nodes. Double arcs are transferring time. Besides, dotted arcs for arrivals and dashed arcs for departures.

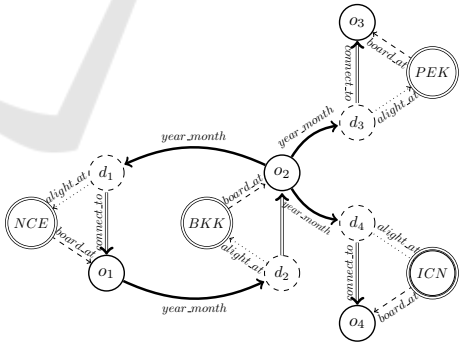

Figure 2: The CFN of the motivating example.

3.2 Graph Database

The CFN is stored in `Neo4j` graph database (Neo Technology, 2017). `Neo4j` is used for many applications, typically recommendation systems and complex networks like transportation network. `Neo4j` graph database follows the property graph model to store and manage its data. In `Neo4j`, data are represented in nodes, relationships, and properties or at-

tributes. Both nodes and relationships contain properties (Robinson et al., 2015). A relationship connects a pair of nodes: start & end, it has a direction and a type. Neo4j proposed *APOC* (Awesome Procedures on Cypher) as stored procedures that regroup a list of procedures (Neo Technology, 2017). In addition, the graph database offers the possibility to implement algorithms as user defined procedures. It simplifies the querying process.

Shortest Path Algorithms in a Graph Database. The Neo4j graph database is proved suited for the shortest path calculation for transport purposes (Miler et al., 2014). The authors compare the performance of Dijkstra implemented on Neo4j and PostgreSQL. Results show that Neo4j outperforms the relational database. It comes from the structure of the graph database, the nodes and relationships are stored in a very compact, quick-to-access format. Once a node or relationship is retrieved, getting the adjacent relationships or nodes is very fast. However, it stores each node's (and relationship's) properties separately, meaning that looking through properties is relatively slow. Then, accessing to properties file are costly which leads to extra IO. This is the important difference between the graph theory and the graph database.

Existing Database. The CFN, stored in the graph database, is generated from a real-world database which is a NoSQL database. This database uses a MongoDB that stores data in a disconnected way and does not use a graph structure. In MongoDB, data are collected and queried monthly. Then it makes sense to create a relationship per period which is a month of the year. The relationship represents flight information (see Figure 2). We dispose of historical data about the last fifteen years. The CFN is generated for two years and has 13,732 nodes and 1,148,303 relationships.

4 PROBLEM FORMULATION

This section gives the formulation of the flight radius problem and also some of its properties.

4.1 Formulation

The flight radius problem consists in retrieving only relevant routes regarding a specific flight, and satisfying the regret constraint. The considered flight is defined by and OD pair and represented by an arc (o,d) in the CFN where $o,d \in V$. Then, traveling from $o_1 \in V$ to $d_1 \in V$ by passing through the arc (o,d) is interesting if and only if the path $\{o_1,..,o,d,..,d_1\}$ between o_1 and d_1 satisfies the regret constraint. The satisfaction of the constraint depends on the shortest path between o_1 and d_1. Let R be a Boolean regret constraint defined on paths of the graph G. Therefore, the problem consists in finding a maximal subgraph, in terms of nodes, such that each node supports a path that satisfies the regret constraint R. It means that there exists a path connecting nodes of this subgraph passing through the arc (o,d) and satisfying one of the criteria. Such paths are called valid paths. The problem is formulated as follows:

Input: *a graph $G=(V,E)$, an arc (o,d), and a regret constraint R*
Output: *a maximal subgraph $G'=(V',E')$ of G that each node belongs to a path passing through the arc (o,d) and satisfying the regret constraint.*

The regret constraint R is defined as follows: let $w(i,j)$ be the weight of the arc (i,j), $l^\star(i,j)$ the length of the shortest path from i to j, and $l(i,j)$ the length of a path passing through the arc (o,d). Then, let consider the following regret constraint which is defined for each criterion:

$$R_{od}(i,j) = l(i,j) \leq l^\star(i,j) + K \qquad (1)$$

Where $K \geq 0$. Each node in the maximal subgraph G' must support at least a valid path satisfying at least one criterion. The definition of the regret constraint can appear similar to the notion of upper tolerance (Shier and Witzgall, 1980). However, it is not the case since our regret function depends on the given regret parameter K.

4.2 Properties

The following property is similar to the one saying that a subpath of the shortest path is also a shortest path (Ahuja et al., 1993).

Property. The subpath of a valid path is also a valid path.

$$l(i,j) \geq l^\star(i,o) + w(o,d) + l^\star(d,j) \qquad (2)$$

The shortest path satisfies the triangle inequality property: following the shortest path from i to o, passing by the arc (o,d), and then following the shortest path from d to j, is always a valid path if one exists.

Proof.

$$
\begin{aligned}
l^\star(i,o) + w(o,d) + l^\star(d,j) &\leq l^\star(i,j) + K \\
\cancel{l^\star(i,o)} + w(o,d) + l^\star(d,j) &\leq \cancel{l^\star(i,o)} + l^\star(o,j) + K \\
w(o,d) + l^\star(d,j) &\leq l^\star(o,j) + K \\
\overrightarrow{R_{od}}(j) = l(o,j) &\leq l^\star(o,j) + K \qquad (3)
\end{aligned}
$$

Where $\overrightarrow{R}$ refers to outgoing direction. Following to the inequality 4.2, the subpath from o to j of a valid path is also valid. In addition, the subpath from i to d is valid. The proof of the property for incoming direction $\overleftarrow{R}$ is symmetric.

Finally, the search can be restricted to the valid shortest paths starting from o or ending at d. Note that when $K = 0$, the set of valid paths represents all the shortest paths passing by the arc (o,d).

Lemma 1. *Any shortest path passing through the arc (o,d) is also a valid path. Therefore, all its nodes are supported.*

5 ALGORITHMS

In this section, we propose four algorithms. First two algorithms decompose the FRP in shortest path problems and solved them in parallel using either the Dijkstra or the Bellman algorithm. The third algorithm extends the Dijkstra algorithm to avoid useless computations. The fourth algorithm extends the Bellman algorithm to compute all shortest paths for all criteria at once. This last algorithm increases memory requirement since processors access the same memory. Thus, we propose to combine between sequential and parallel processing to optimize the runtime and memory consumption. These algorithms perform parallel computations when it is possible.

input : a CFN $G = (V,E)$, an arc (o,d), the criteria C
output: the set of supported nodes S

For sake of simplicity, the algorithm returns the set of supported nodes and worries about storing the parents of the supported nodes. In practice, the algorithm also returns the union of the supported trees for each direction and for each criterion. Based on the regret functions, let's define:

$$
R(i,dir) = \begin{cases} \overrightarrow{R_{od}}(i), & \text{if } (dir = out) \\ \overleftarrow{R_{od}}(i), & \text{if } (dir = in) \end{cases}
$$

5.1 Shortest Path Decomposition

The shortest path decomposition solves the FRP problem by using two shortest path problems, one from the origin o and the other from the destination d, for each direction and for each criterion in parallel. Then supported nodes are computed by checking the regret constraint.

Algorithm 1: SP Decomposition.

```
foreach dir ∈ {in, out} in parallel
    foreach c ∈ C in parallel
        ShortestPaths(o, c, dir);
        ShortestPaths(d, c, dir);
        foreach i ∈ V do
            if R(i, dir) then
                S ← S ∪ {i} ;
            end
        end
    end
end
return S
```

The shortest path subroutine is either Dijkstra or Bellman algorithm. The computation of the shortest paths from the origin and from the destination is sequential for simplifying the search of supported nodes.

5.2 Flight Radius Algorithms

Here, we design variants of Dijkstra and Bellman algorithm tailored for solving the flight radius problem.

5.2.1 Dijkstra FR Algorithm

We present an algorithm that computes the shortest path from o, and then computes lazily the shortest path from d by skipping unsupported nodes (see algorithm 2). Here, the criteria are processed sequentially. At each iteration, the algorithm scans the node with the minimum weight and then relaxes its neighbors. So, we check if the node satisfies the regret function otherwise we skip the node. Therefore, we skip non supported nodes that cannot be extended into valid paths. The algorithm ends if the queue becomes empty or all shortest paths to the supported nodes are found. Function `SCAN(i,c,dir)` calls the function presented in 2.2 for each criterion and direction.

Algorithm 2: Dijkstra FR Algorithm.

```
foreach dir ∈ {in, out} in parallel
    foreach c ∈ C do
        Dijkstra(o, c, dir);
        enqueue(Q, d);
        while Q ≠ ∅ and isNotClosed() do
            i ← argmin_{j∈Q}(d[j]);
            dequeue(Q, i);
            if R(i, dir) then
                S ← S ∪ {i};
                SCAN (i, c, dir);
            end
        end
    end
end
return S
```

5.2.2 Bellman FR Algorithm

We propose a variant of Bellman algorithm (algorithm 3) that computes at once all shortest paths from the origin and from destination for all criteria. Then, more nodes are added to the queue. Here, the `SCAN` function updates all paths from the origin and from the destination for all criteria.

Algorithm 3: Bellman FR Algorithm.

```
foreach dir ∈ {in, out} in parallel
    enqueue (Q, o);
    enqueue (Q, d);
    while (Q ≠ ∅) do
        i ← dequeue(Q);
        SCAN (i, C, dir);
    end
    foreach i ∈ V do
        if R(i, dir) then
            S ← S ∪ {i};
        end
    end
end
return S
```

6 EXPERIMENTS

In this section, we evaluate the algorithms to solve the FRP on real-world datasets. The SP decomposition algorithms used massive parallelism. However, the FRP algorithms used more optimization than parallelism. In other words, we are especially interested in comparing the runtime of the algorithms which impacts the user experience of `PlanetOptim`. Besides, the number of scanned nodes by the algorithm. This metric is very important since it determines the accessed property values (criteria) of a relation. Those properties are accessed incurs extra IO the first time. The reason is that properties reside in a separate store file from the relationships (after that, however, they are cached) (Robinson et al., 2015). Our experimental protocol aims to answer the following questions: Which algorithm is the fastest and, will, therefore provide the best user experience? Is it worthwhile to design algorithms for flight radius problem compared to the decomposition into shortest path problems? Does reduction of the number of scanned nodes lead to a reduction of the runtime? How does the performance evolve with the number of criteria? First, we present how the benchmarks instances have been generated. Second, runtime of the algorithms are compared, and then, the relation between the runtime and the number of scanned nodes is studied. Last, we analyze the ratio of the flight radius algorithms over shortest path decompositions depending on the number of criteria. All the experiments were led on a computer running on Ubuntu 16.04.5 with 32 GB of RAM and one Intel Core i7-3930K 3.20GHz processor (6 cores). The implementation is based on `Neo4j` and *APOC* version 3.2.0. All algorithms are implemented in Java 8.

6.1 Instances Generation

The CFN contains historical data for the years 2016 and 2017 (24 year-months). Each year-month corresponds to a different graph. The condensed flight network contains 13,732 nodes and 1,148,303 arcs. A benchmark instance must specify the year-month, the OD-pair (o,d), the number of criteria, and their regrets.

The number of criteria is one (time), two (time, distance), or three (time, distance, cost). For each criterion, two values are considered for its regret K: 0 and the median (over all relations and year-months). The value 0 means that only shortest paths passing through the arc (o,d) are valid, whereas many other paths are valid with the median. For instance, the median duration of a flight is approximately two hours. So a path between i and j passing through the arc (o,d) is valid if it does not exceed the duration of the shortest path between i and j by four hours (the median duration plus the minimum connection time).

For each year-month, each number of criteria, and each combination of criteria values, a few pairs of origin and destination (o,d) are drawn randomly. At the end, more than ten thousand instances have been tested.

Table 1: Distribution of runtimes in milliseconds.

	SP Decomposition		FR Algorithm	
	Dijkstra	Bellman	Dijkstra	Bellman
avg	266	500	231	604
std	60	198	90	282
max	418	1328	644	1612

6.2 Algorithms Comparison

Table 1 gives the average, standard deviation, and maximum runtime in milliseconds of the shortest path decompositions and of the flight radius algorithms using Bellman or Dijkstra. The Dijkstra variants are approximately two times faster and have a lower standard deviation than their Bellman counterparts. The algorithm based on Dijkstra is slightly faster than the decomposition whereas it is not the case for Bellman.

Figure 3 analyzes the relation between the runtime and the number of scanned nodes for each algorithm. Each point represents one instance and its x-coordinate is the runtime in milliseconds, whereas its y-coordinate is the number of scanned nodes. The color of a point indicates which algorithm solved the instance. The flight radius algorithm based on Bellman scans fewer nodes than the shortest path decomposition based on Bellman (red points are below the blue ones), but without reducing the runtimes (blue points are on the left of the red ones).

It means that the additional time spent to read all properties is not compensated by the decrease in the number of scanned nodes. For Dijkstra, the number of scanned nodes for the decomposition only depends on the number of criteria whereas it is not the case

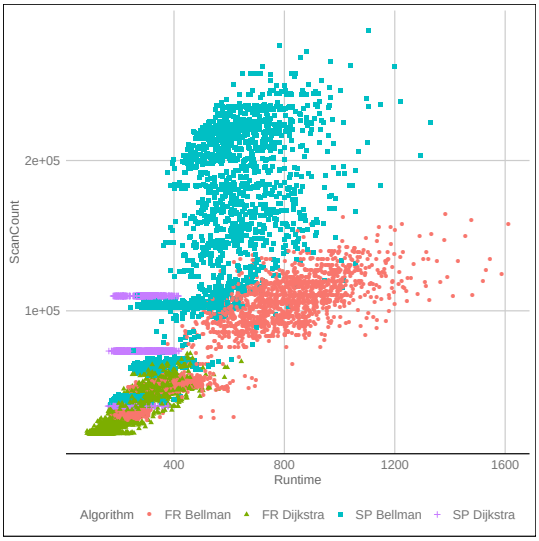

Figure 3: Analysis of the runtimes and scan counts.

Table 2: Improvements of the FR algorithms over the SP decompositions.

	Dijkstra		Bellman	
#Criteria	Runtime	#Scans	Runtime	#Scans
1	0.69	0.57	1.04	0.81
2	0.79	0.41	1.15	0.57
3	1.09	0.43	1.37	0.49

for the flight radius algorithm. In this case, a lower number of scanned nodes implies a reduction of the runtime because the number of red properties is also reduced (green points are on the left and below purple points). As expected, Dijkstra based algorithms scan fewer nodes than those based on Bellman.

Last, Table 2 gives the geometric mean of the improvements provided by FR algorithms over the SP decompositions in terms of runtimes and number of scanned nodes. The improvement is the ratio of the runtime of FR algorithm over the SP decomposition for Bellman or Dijkstra. The improvement is lower than 1 if the FR algorithm is better than the SP decomposition and greater than 1 otherwise. Both FR algorithms based on Dijkstra or Bellman reduce the number of scanned nodes and the reduction increases with the number of criteria. The runtimes are not reduced for Bellman, but are also reduced for Dijkstra when the number of criteria is one or two. When there are three criteria, the reduction of scanned nodes does not compensate for the additional parallelization of the decomposition.

To conclude, FR algorithm based on Dijkstra is the most efficient, and satisfies the real-time constraint of `PlanetOptim`. Besides, the runtimes of the algorithms increase with the number of criteria in spite of the parallelization. When there are three criteria, the reduction of the number of scanned nodes does not help to reduce the runtimes of FR algorithm based on Dijkstra, a perspective is to parallelize along the criteria.

7 CONCLUSION

This work presents some algorithms for solving the FRP on the CFN. The problem was formulated as finding a maximal subgraph, in terms of nodes, such that each node supports a path satisfying a regret constraint. This constraint is used to model passengers preferences: time, cost, or distance. We propose four algorithms based on SPA to solve the FRP. These algorithms combine sequential and parallel processing to overcome time and memory costs. We are working on an industrial context: the CFN is generated from

a *Milanamos* database and stored in a graph database. Implementing SPA in graph databases is challenging since accessing to properties files, where criteria information are stored, is costly. This is the important difference between graph theory and graph database. Then, two metrics are used to evaluate algorithm's performances: runtime and number of scanned nodes. The last one metric evaluates the number of properties file access. Experiments show that the FR algorithm based on Dijkstra is the most efficient, and satisfies the real-time constraint of `PlanetOptim`. The next step of the study is to include QSI models to speed up computations. Latter, our solution will be integrated in the industrial application `PlanetOptim`.

ACKNOWLEDGMENTS

We would like to thank Carine Fédèle for her insightful comments on the paper, as these comments led us to an improvement of the work.

REFERENCES

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network flows: theory, algorithms, and applications*. Prentice hall.

Cherkassky, B. V., Goldberg, A. V., and Radzik, T. (1996). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174.

Delling, D., Pajor, T., Wagner, D., and Zaroliagis, C. (2009). Efficient Route Planning in Flight Networks. *ATMOS*, 12.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271.

Jacobs, T. L., Garrow, L. A., Lohatepanont, M., Koppelman, F. S., Coldren, G. M., and Purnomo, H. (2012). Airline planning and schedule development. In *Quantitative Problem Solving Methods in the Airline Industry*, pages 35–99. Springer.

Miler, M., Medak, D., and Odobašić, D. (2014). The shortest path algorithm performance comparison in graph and relational database on a transportation network. *Promet-Traffic&Transportation*, 26(1):75–82.

Neo Technology (2017). Neo4j. `https://www.neo4j.com`.

Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

Shier, D. R. and Witzgall, C. (1980). Arc tolerances in shortest path and network flow problems. *Networks*, 10(4):277–291.