Flight Planning in Free Route Airspaces*

Casper Kehlet Jensen[1], Marco Chiarandini[2], and Kim S. Larsen[3]

1 **University of Southern Denmark, Odense, Denmark**
`casper@snemanden.com`
2 **University of Southern Denmark, Odense, Denmark**
`marco@imada.sdu.dk`
3 **University of Southern Denmark, Odense, Denmark**
`kslarsen@imada.sdu.dk`

Abstract

We consider the problem of finding cheapest flight routes through free route airspaces in a 2D setting. We subdivide the airspace into regions determined by a Voronoi subdivision around the points from a weather forecast. This gives rise to a regular grid of rectangular regions (quads) with every quad having an associated vector-weight that represents the wind magnitude and direction. Finding a cheapest path in this setting corresponds to finding a piece-wise linear path determined by points on the boundaries of the quads. In our solution approach, we discretize such boundaries by introducing border points and only consider segments connecting border points belonging to the same quad. While classic shortest path graph algorithms are available and applicable to the graphs originating from these border points, we design an algorithm that exploits the geometric structure of our scenario and show that this algorithm is more efficient in practice than classic graph-based algorithms. In particular, it scales better with the number of quads in the subdivision of the airspace, making it possible to find more accurate routes or to solve larger problems.

1998 ACM Subject Classification F.2.2 [Nonnumerical Algorithms and Problems] Geometrical Problems and Computations, G.2.2 [Graph Theory] Path and Circuit Problems

Keywords and phrases Flight planning, Geometric shortest path, Free route airspace, Vector weighted paths, Vector weighted planar subdivisions

Digital Object Identifier 10.4230/OASIcs.ATMOS.2017.14

1 Introduction

In flight planning, users seek the cheapest flying route between departure and arrival airports. The cost of a route is determined by a function of fuel consumed and flying time. These measures are related as they depend on weather conditions, which are dynamic. The flight routes are calculated through a network of waypoints and airways representing a discretization of the 3D airspace. Additionally, different types of constraints are imposed on these routes by a central control institution. For example, there are constraints with the purpose of regulating traffic or avoiding specific airspaces. These constraints can be updated several times a day. Therefore, flight routes are planned only few hours before the flight and efficient algorithms are needed. The particular cost dependencies and constraints make the problem more challenging than classic shortest path problems [3, 14, 13].

* The third author was supported in part by the Danish Council for Independent Research, Natural Sciences, grants DFF-1323-00247 and DFF-7014-00041.

A *free route airspace* (FRA) is a specified airspace within which users can freely plan a route between a defined entry point and a defined exit point, with the possibility of routing via intermediate (published or unpublished) waypoints. In this case, we face the problem of finding a best path through a continuous airspace between two endpoints. However, because of wind conditions, the best path is *not* in general a straight line and hence not trivially determined – see for example [2], providing an example of a wind-optimal route from Jakarta to Honolulu that is 11% longer than the direct great circle route, but 2% faster and uses 3% less fuel. Usage of the jet stream for flights over the Atlantic is also a common example of this (see e.g. [11]).

The motivation for introducing free route airspaces is to try to overcome some of the challenges in aviation, such as efficiency, capacity, and even environmental problems. According to Eurocontrol, the European Organization for the Safety of Air Navigation [9], using FRAs has the potential to reduce flying distances by approximately 7.5 million nautical miles per year, representing a $45,000$ tons fuel save, or a reduction of emissions of $150,000$ tons. Also, FRAs offer fewer conflicts in terms of aircraft collision avoidance, as aircrafts are more evenly spread out across the airspace compared to the current fixed route network.

Currently, FRAs can arise as part of the more common airways network, but for the reasons above, they are expected to become more widespread in the future. The planning tools to deal with them do not exploit the full potential and rely on several heuristics. For example, a common procedure is to introduce arbitrary intermediate airway points in the FRA and to consider all possible airways between them and the entry/exit points. Beside not finding real optimal paths, such an approach is also computationally demanding.

In this work, we investigate the problem of finding an optimal path through a free route airspace between two entry/exit points, taking wind conditions into account. We propose a data structure for discretizing the airspace into regions of constant wind and present a geometric algorithm to solve the problem in this discretized environment. We show that our algorithm is able to exploit wind conditions and find near-optimal solutions for the given weather forecast. We perform an experimental study of the scaling of the running time with respect to the number of regions that subdivide the airspace and show that our algorithm scales more efficiently in comparison with more classic graph-based approaches that find the same solutions. In an extension of this work, we design heuristics to join adjacent regions with similar wind conditions and show that our algorithm exhibits better behavior than classic approaches also with respect to the trade-off of accuracy vs. efficiency that these heuristics allow to explore.

According to [4, 5], the problem of finding a geometric shortest path on the plane with polygonal obstacles is solvable in $O(n \log(n))$, where n is the total obstacle vertices. This run time has been shown to be optimal. On the other hand, the same problem in the 3D space with polyhedral obstacles is NP-hard and has attracted research for approximation algorithms.

A few versions of the problem of finding shortest path queries in a continuous environment with weighted regions have been studied. For example, in [12], a particular case is investigated in which the cost of a continuous path is determined by a cost-weighting function that depends on the position and the direction of motion. The authors approach the problem by only considering paths formed by concatenating straight-line segments for a suitable discretization of the environment. This work is aimed at unmanned air vehicles (for military purposes) for finding best low-risk paths in an environment with hostile ground defenses that should be avoided.

Another work [15] considers shortest path problems on the plane with a weighted polygonal subdivision. This algorithm applies the "continuous Dijkstra" paradigm and exploits a

physical property of such paths (that they obey Snell's law of refraction). However, the complexity of the algorithm does not seem promising since the best bound given is $\mathcal{O}(n^8 L)$, where n is the number of vertices of the subdivision and L some parameter that specifies the precision of the problem instance and linked to the desired approximation margin.

An improvement of [15] is given in [1]; these authors present an approximation scheme (computing paths whose costs are within a factor of $1+\epsilon$ of the shortest paths costs, for arbitrary $\epsilon > 0$) for the problem of finding shortest paths on a polyhedron consisting of n convex faces and each face with a positive non-zero real valued weight. Their algorithm runs in $\mathcal{O}\left(\frac{n}{\epsilon}\log(\frac{1}{\epsilon})(\frac{1}{\sqrt{\epsilon}} + \log(n))\right)$.

We found inspiration for our work in the approach described in [6] and [16]. In these works, the authors represent the environment using quad-trees (for 2D) and oct-trees (for 3D), that is, tree based data structures whose leaves represent regions of constant weight. Each region at the leaves of the tree is framed with smaller square cells that determine the granularity of the search space. Further, the authors propose an efficient geometric algorithm for finding a shortest path between the smaller cells on the frame of the quad-tree leaves, running in $\mathcal{O}(P\log(P)) + \mathcal{O}(T_{\text{neighbor}})$, where P is the total number of cells in the discretization and T_{neighbor} is the time to update neighbor pointers in the quad-tree data structure. They provide a (somewhat natural) extension to 3D, in which they propose an algorithm that runs in $\mathcal{O}\left(n^2\log(m)\right)$, where n is the number of cells on a side in a region (if the region was filled, it would contain n^3 cells) and $m \leq P$ the size of the heap. These algorithms are a factor n better than previous grid based approaches. However, the 3D algorithm is only able to compute paths w.r.t. metrics L_1 and L_∞ (that is, not L_2). In this paper, we reconsider those approaches by introducing border points rather than border cells, which we believe should make the algorithm simpler. Moreover, all the methods mentioned assume real number weights on the regions while in our case we need to consider vector weights, since the wind has both a magnitude and a direction. A main contribution here is the generalization of the geometric arguments from [6] to vector weights, resulting in a more efficient search space examination.

Due to space constraints, many details are omitted in this conference version. Proofs of all results, claims and theorems have been established and these will be given in a full version of the paper.

2 Preliminaries

A FRA is in general a subset of the atmosphere of Earth. Here we assume that a FRA is the 3D space between a top and bottom face, both defined by a single 2D polygon. Such a polygon, denoted by F, can be described by the set of its vertices, $F = \{\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_k\}$, where $\vec{p}_i = (\phi, \psi)$, $i = 1, \ldots, k$, ϕ being the latitude and ψ being the longitude. On the border of a FRA, there are special waypoints that we call entry/exit points.

We restrict ourselves to FRAs at a fixed altitude and thus consider FRAs that are 2D polygons. Neglecting the third dimension simplifies our approach but keeps our solutions relevant in practical terms. Indeed, a typical way of looking at flight routes is by decomposing them into their 2D projections on the surface of the Earth and their vertical profiles [14, 3]. The vertical profile is commonly decomposed into three phases: climbing, cruising, and descending.[1] The cruising phase is the dominant, and takes place at an altitude that is

[1] See `http://flightaware.com/` or `http://flightradar24.com`, for example.

favorable for the performance of the aircraft. Therefore, in most situations, the flight has to cross FRAs in the cruising phase, where the airplane has reached its desired altitude and a 2D solution is sufficient. Alternatively, in situations where one of the end points of a route lies inside an FRA, the best 2D path through it may still be used to plan a 3D path.

The weather forecast service may be provided by the World Area Forecast System (WAFS) [17]. Wind data on Earth is available for each 1.25th degree of latitude and 1.25th degree of longitude and for several altitudes, yielding a 3D grid of geodesic points. For each such point, *wind barbs* are available, providing a visual description of the wind speed and direction. In mathematical terms, they define *wind vectors*. This weather data is updated at regular intervals of 6 hours, with a resolution of 3 hours. In our 2D simplification, we consider a 2D grid of points, W, corresponding to one single level of the 3D grid given by a desired altitude.[2]

Locations of points in a three dimensional spherical space can be described by means of Cartesian coordinates on the plane by applying a Mercator projection. In such a projection of the world map, lines of constant latitude and longitude become straight horizontal and vertical lines, respectively. Hence, once projected to the Cartesian plane, the 2D weather grid becomes a regular, rectangular grid.

Wind conditions in other locations than those provided by the grid W can be inferred by interpolation; by linear regression, for example. Here, we prefer another approach and assume that every point in the Cartesian plane has the same wind vector as the grid point of W at the smallest Euclidean distance. Denoting by $\vec{p}_{i,j}$ the location of the point $(i,j) \in W$ and by $\vec{w}_{i,j}$ the associated wind vector, we can determine the regions $R_{i,j} \subseteq \mathbb{R}^2$ with constant wind vector $\vec{w}_{i,j}$ as follows: $R_{i,j} = \{\vec{p} \in \mathbb{R}^2 \mid \forall\,(r,s) \in W, (r,s) \neq (i,j)\colon\ \|\vec{p} - \vec{p}_{i,j}\| \leq \|\vec{p} - \vec{p}_{r,s}\|\}$. This yields a Voronoi tessellation (partitioning) of the plane (see e.g. [8]), which preserves the regular, rectangular structure of W.

In what follows for a polygon F representing a FRA, we assume that we also have a rectangular tessellation $\mathcal{Q}$ of F consisting of rectangular regions (quads) Q_i, $i = 1, \ldots, K$ and $\mathcal{Q} = Q_1 \cup Q_2 \cup \ldots \cup Q_K$ such that all points of a region have the same wind vector $\vec{w}_i$. It is easy to think of this rectangular tessellation as the same Voronoi tessellation derived from the wind grid. However, it can also be a more general rectangular tessellation, in which several regions of similar wind conditions are joined together; see Fig. 1.

Wind-dependent travel time function. Consider two points $\vec{p}, \vec{q} \in \mathbb{R}^2$ of a region of constant wind $\vec{w}$, and let $\vec{p}$ be the position of the aircraft and $\vec{q}$ its desired destination. With respect to the ground, the aircraft travels along the direction $\overrightarrow{pq} = \vec{q} - \vec{p}$, where $\|\overrightarrow{pq}\|$ is the *ground distance* between $\vec{p}$ and $\vec{q}$. When the aircraft flies through the air, it is affected by the wind; see Fig. 2. To hit the goal $\vec{q}$, it has to head in a proper direction, possibly different from $\overrightarrow{pq}$. This direction, relative to the atmosphere and hence called *air velocity*, is denoted by $\vec{v}_A$ and $\|\vec{v}_A\| = h$ is the *true airspeed*. The ground velocity $\vec{v}_G$ in the direction $\overrightarrow{pq}$ is given by: $\vec{v}_G = \vec{v}_A + \vec{w}$.

Expressing the wind vector in polar coordinates $\vec{w} = (\omega, \theta)$, where ω is the wind magnitude and θ the angle between $\vec{w}$ and $\overrightarrow{pq}$, and using the laws of trigonometry, we can derive the

[2] Latitude degrees are defined from 90 deg north to 90 deg south and longitude degrees from 180 deg west to 180 deg east. Thus, real-life 2D weather data is given for a 2D grid of 144×288 points, i.e., $W = \{(i,j) \mid i = 0, 1, ..., 143$ and $j = 0, 1, ..., 287\}$.

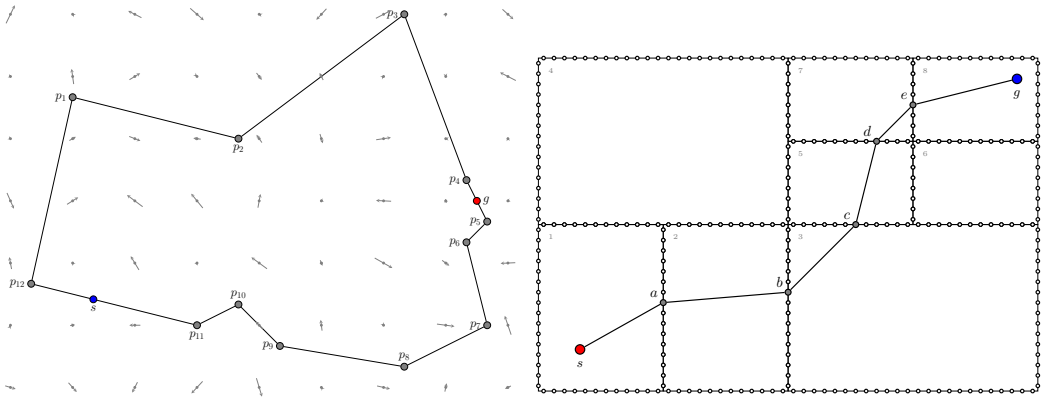

Figure 1 Left: a polygon defined by points p_i, $i = 1, ..., 12$, representing a (fictional) free route airspace. Our problem is to find the best path between a source s and a goal g in terms of travel time, which depends on the wind conditions, represented by wind vectors indicating direction and magnitude at evenly spaced locations. Right: A path from s to g in a tessellation of a flying area (different from the left one). Border points are emphasized by circles.

travel time for the aircraft to go from $\vec{p}$ to $\vec{q}$. That is,

$$t_{\vec{p},\vec{q}}(\vec{w}) = \frac{\|\overrightarrow{pq}\|}{\|\vec{v}_G\|} = \frac{\|\overrightarrow{pq}\|}{\omega\cos\theta + \sqrt{h^2 - \omega^2\sin^2\theta}}. \tag{1}$$

Under the reasonable assumption that the true airspeed h is larger than the wind speed ω, then $h^2 - \omega^2\sin^2\theta > 0$ and Eq. (1) gives a valid value.

3 Algorithms for Cheapest Path through Vector-Weighted Regions

We discretize the flight route problem to obtain a graph representation. This makes it possible to apply classic Dijkstra and $A^\star$ algorithms directly. Then, we define our new algorithm, which uses the same search graph, but exploits geometric properties of the environment it models to reduce the run-time. The algorithms we present are optimal up to the approximation due to the discretization. Intuitively, we then approach the continuous optimal route when the number of border points increase.

We assume that we have a 2D polygon F representing a FRA at a specific flight level, and two additional points, s and g, representing a source and a goal, respectively. Further, we assume that we have a tessellation (partition) $\mathcal{Q}$ of the polygon F in rectangular regions (quads) $Q_1, \ldots, Q_K$, and vector weights $\vec{w}_1, \ldots, \vec{w}_K$ associated with each quad, representing the direction and intensity of the wind. The situation described is sketched in Fig. 1, left.

A continuous curve in F is a path P and its cost is $c_P = \sum_{i=1}^{n} t_{P_i}(\vec{w}_i)$, where $t_{P_i}(\vec{w}_i)$ denotes the travel time of the intersection of P with the quad Q_i, that is, $P_i = P \cap Q_i$. Given two distinct points s and g in P, a minimum cost (s, g)-path joining s and g is called a geometric shortest path. With constant vector weights associated with each quad, geometric shortest paths are simple (non-self-intersecting) and consist of a sequence of linear segments. We define *linear paths* to be simple paths consisting of quad-crossings exclusively. A linear path P is made by a sequence of segments or, equivalently, as a sequence of the endpoints $a_0, \ldots, a_\ell$ of these segments.

In order to search for such paths, we discretize the borders of quads into a set of *border points*. For a quad $Q \in \mathcal{Q}$, we refer to its four sides from the perspective of an entering path

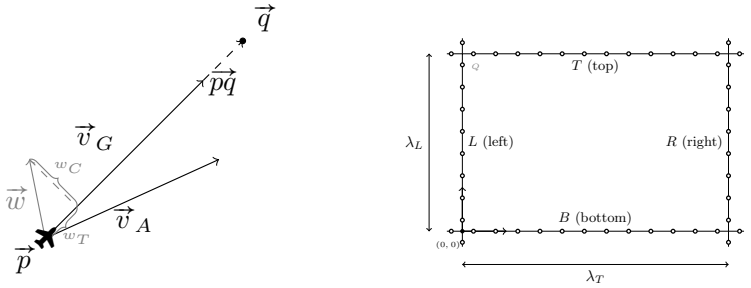

Figure 2 Left: an aircraft traveling along $\overrightarrow{pq}$ with ground velocity $\vec{v}_G$ affected by wind vector $\vec{w}$ travels with true air velocity $\vec{v}_A$. The individual components of the wind vector, $w_C = \omega \cdot \sin\theta$ and $w_T = \omega \cdot \cos\theta$ denote the *crosswind* and *tailwind* components, respectively. The ground speed $\|\vec{v}_G\|$ is given by $w_T + \sqrt{\|\vec{v}_A\|^2 - w_C^2}$. Right: a quad Q with border points (i, S), $S \in \{B, T, L, R\}$ and the local coordinate system.

as bottom (B), opposite (or top) (T), left (L) and right (R). The definition is relative to a point on the side of Q that is denoted as B. We also define a local coordinate system with its origin $(0,0)_Q$ at the bottom (B) left (L) corner of the quad. We decorate the sides $S \in \{B, T, L, R\}$ with a set of equi-distant vertex points. Let k_S be the number of points on side S and let $k_S = m$ if $S = \{L, R\}$ and $k_S = n$ if $S = \{B, T\}$. Let also λ_S be the length of a side $S \in \{B, T, L, R\}$, with $\lambda_L = \lambda_R$ and $\lambda_T = \lambda_B$. The border points are placed at a distance $\epsilon_S = \lambda_S / k_S$ apart. For example, the coordinates with respect to the local system of the points on the side L are $(\epsilon_S/2, 0)_Q, (\epsilon_S(1/2+1), 0)_Q, \ldots, (\epsilon_S(1/2+k_S-1), 0)_Q$. There are no border points at the corners of the quads. We represent a border point $p \in Q$ by the tuple $(i, S)_Q$, where $i \in \{0, 1, ..., k_S\}$ is the index of the point on side $S \in \{B, T, L, R\}$; see Figure 2.[3]

For each quad $Q \in \mathcal{Q}$, we let an adjacent quad be any quad that shares (part of) a border with Q. Thus, $\mathcal{Q}$ is essentially implemented as a planar graph with an adjacency list for each quad. Hence, the amount of memory required to store the tessellation and the border points is $O(Kn)$, where we assume $n = m$. If the tessellation is derived from the original weather grid, then it is a regular grid, where each quad can be identified by two indices (i, j), and neighboring quads can be derived by incrementing or decrementing the indices, and adjacency lists are not needed.

We indicate by A_Q the set of border points on the four sides of $Q \in \mathcal{Q}$ and by A_Q^S the border points on each single side $S \in \{B, T, L, R\}$ of Q. Border points on a common side of two quads $Q, Q' \in \mathcal{Q}$ belong to both A_Q and $A_{Q'}$. Let Q_s and Q_g denote the quads containing s and g, respectively. We limit the reachability of border points of a side of a quad to points on one of the other three sides and the two closest points to the left and right on their own side. As a special case, a most extreme border point on a side can also reach the neighboring point on the side which is a continuation of its own side (in the neighboring quad). Thus, the set of reachable points $R(p)$ from $p = (i, 0)_Q \in Q$ is $R(p) = \bigcup_{S \in \{T,L,R\}} A_Q^S \cup (\{(i-1, 0)_Q, (i+1, 0)_Q\} \cap A_Q^B)$.

For the tessellation $\mathcal{Q}$, we define the graph $G_{sg} = (V, E)$, where vertices in V represent border points and edges in E represent links between reachable border points. Formally,

[3] Sometimes we represent a border point p also by means of the local coordinates of the quad to which it belongs; although, in this case, it would be more appropriate to denote p as a vector we continue to denote the point simply by p.

$V = \bigcup_{Q \in \mathcal{Q}} A_Q$ and $E = \{uv \mid v \in R(u), u \in A_Q, Q \in \mathcal{Q} \setminus \{Q_s, Q_g\}\} \cup \{sv \mid v \in A_{Q_s}\} \cup \{ug \mid u \in A_{Q_g}\}$; see Figure 3, showing a tessellation $\mathcal{Q}$ with four quads and the corresponding graph G_{sg}. The cost c_{uv} associated with an edge $uv \in E$ is the travel time between the corresponding border points $p_u, p_v \in A_Q$, that is, $t_{p_u,p_v}(\vec{w}_Q)$.[4]

For a border point p, let $\phi(p) = (s = a_0, a_1, a_2, \ldots, a_{k-1}, a_k = p)$ be the cheapest path from s to p, across k border points $a_i \in Q_i \in \mathcal{Q}$ for $i = 2, \ldots, k$ and $a_1 \in Q_s$ as the first border point on the path from s to p. To each border point $p \in Q$, we associate the following data: the travel time t_p from s to p along path $\phi(p)$ defined by $c_p = \sum_{i=0}^{k-1} c_{a_i a_{i+1}}$ and a pointer α_p to p's predecessor in $\phi(p)$, hence $\alpha_p = a_{k-1}$.

Initially, each border point is assigned a large value to t_p and a null pointer to α_p. We say that a border point $u \in Q$ *covers* another border point $v \in Q$ when a path ending at u is extended to v and the values t_v and α_v are, consequently, updated. Once the initial values t_p and α_p of a border point p have been updated, we say that the border point p is *covered*. A border point $p \in A_Q$ that has been covered by a point u not in Q is said to be an *entry point for* Q. Points covered by points on the same side are not categorized as entry points.

Graph-based approaches. We can find shortest paths in G by applying classic Dijkstra [7] and $A^\star$ algorithms. These algorithms expand records of vertices from an open list $\mathcal{H}$ until a path from source to goal is proven optimal. The expansion of a vertex v amounts to inserting all vertices reachable from v into the heap $\mathcal{H}$, or updating their keys if they are already in $\mathcal{H}$. When extracting a vertex $v \in V$ from the open list, priority is given to the one of smallest cost (best-first) or smallest sum of the cost of the path to v and a heuristic estimate $h(v)$ of the cost from the corresponding node to the goal (in case of $A^\star$). The algorithm terminates when the goal g has been reached and the incumbent best path to g is cheaper than the cheapest record in $\mathcal{H}$. Using best-first, the solution returned is optimal. For $A^\star$, the solution returned is optimal if the heuristic is *admissible* (it never overestimates the cost from v to the goal). If the heuristic is also *consistent* (for each edge $uv \in E$, $h(u) \leq c_{uv} + h(v)$), then we only need to expand a node once.

For any vertex $v \in V$ corresponding to a border point p_v, we define the heuristic value $h(p_v)$ to be the travel time needed to traverse the straight line distance with a tail wind of intensity equal to the largest in the FRAs. Formally, $h(p_v) := t_{p_v g}(\vec{w}'_{p_v g})$, where $\vec{w}'_{p_v g} := \frac{\overrightarrow{p_v g}}{\|\overrightarrow{p_v g}\|}\omega_{\max}$ and $\omega_{\max} = \max_{Q \in \mathcal{Q}} \|\vec{w}_Q\|$. This heuristic is admissible and consistent.

We have implemented the open list $\mathcal{H}$ as a min-heap. For a vertex $v \in V$, the corresponding point $p_v = (i, S)$, $S \in \{B, T, L, R\}$, $i = 1, \ldots, k_S$, and α_{p_v} are stored with the key t_p. Smaller keys indicate higher priority in the list and the heap supports all the usual operations needed for the implementation of Dijkstra's algorithm. The use of $\mathcal{H}$ thus defined leads to an asymptotic running time for Dijkstra of $\mathcal{O}((V + E)\log(V))$ or, in other terms, since each of the K A_Q contains $O(n)$ points and $O(n^2)$ edges, $O(Kn^2 \log(Kn))$.[5]

Geometric algorithm. The main idea is to speed up node expansion by avoiding to expand to border points that can be reached more conveniently from other nodes already in the

[4] Note that the definition of cost for edges between vertices representing points on the same side is ambiguous because we defined border points to belong to both quads Q and Q' sharing the side. We break this ambiguity in the search algorithms by always considering the wind vector of the quad Q in which u is an entry point. An alternative choice would have been to consider on that segment the average of the wind vectors in the two adjacent quads. However, this choice would break the triangular inequality and require more computation.

[5] Using Fibonacci heaps [10], this could be brought down to $O(Kn\log(Kn) + Kn^2)$.

1 **Function** GeomBestFirst($\mathcal{Q}$, s, g)
2 **foreach** quad $Q \in \mathcal{Q}$ **do**
3 **foreach** point $p \in A_Q$ **do**
4 $t_p \leftarrow \infty$, $\alpha_p \leftarrow$ NIL
5 $\mathcal{H}$ = new empty priority queue
6 Find quad $Q_s \in \mathcal{Q}$ that contains s and $Q_g \in \mathcal{Q}$ that contains g
7 $D_s = A_{Q_s}$
8 $\beta_s = \arg\min\{c_{sp} \mid p \in D_s\}$
9 $\mathcal{H}$.Insert($c_{\beta_s,s}$, s)
10 **while** not $\mathcal{H}$.IsEmpty() and not all points in A_{Q_g} are covered **do**
11 $\gamma_r, r = \mathcal{H}$.ExtractMin() ▷ Smallest key and associated point
12 $t_{\beta_r} = \gamma_r$; $\alpha_{\beta_r} = r$ ▷ Cover β_r
13 **foreach** $S \in \{B, T, L, R\}$ **do**
14 $D^S_{\beta_r}, p_S =$ CompeteOnSide(β_r, S)
15 $\beta_{\beta_r} = \arg\min\{t_{\beta_r} + c_{\beta_r,p_S} \mid S \in \{B, T, L, R\}\}$
16 $\mathcal{H}$.Insert($t_{\beta_r} + c_{\beta_r,\beta_{\beta_r}}, \beta_r$) ▷ Smallest new key and associated point
17 **if** D_r is not empty **then**
18 $\beta_r = \arg\min\{t_r + c_{r,p} \mid p \in D_r\}$
19 $\mathcal{H}$.Insert($t_r + c_{r,\beta_r}$, r)
20 **return** $\phi(g)$

Algorithm 1: Main path finding algorithm.

open list. This can be achieved by exploiting the geometry of the environment and the vector weights. An overview of the algorithm is given in Alg. 1; many of the ingredients are described below. Differently from the previous Dijkstra and $A^\star$ algorithms, the key of an element $v \in V$ stored in the open list $\mathcal{H}$ is not the cost of the path from source to v, but the cost of the path from the source, through v, to v's cheapest reachable successor node.

To every border point $u \in A_Q$, $Q \in \mathcal{Q}$, we associate a *domain* set D_u, which is a subset of uncovered border points in A_Q that the point may try to cover in the future. $D^S_u \subseteq D_u$ is the domain set restricted to points on side S. The set D^S_u is a connected interval of points, which can be represented as $D^S_u = [\underline{i}_u, \bar{i}_u]$. The first entry point p that enters a side B of Q will initially have all other border cells in Q as its domain. However, if another entry point of Q on the same side B, q, is extracted from $\mathcal{H}$, then p and q *compete* for their domains. Thus, the border points of Q are partitioned into two subsets such that the domain of q becomes the border points that are less costly to reach from q than from p, and vice versa. We say that an entry point is *active* as long as its domain is not empty. When an entry point is no longer active, we say that it is *blocked*.

For this algorithm, we extend the information maintained at a border point p to contain, in addition to t_p and α_p, the domain of the point and the cheapest reachable node in the same quad, β_p. Note that this is again a point, so the notation β_{β_p} is meaningful.

After the initialization of the data associated with border points (lines 2–4), all nodes $v \in V$ representing border points of Q_s are assigned to the domain of s and s is inserted into $\mathcal{H}$ with the cost to the cheapest reachable point in A_{Q_s} as its key (lines 6–9). For each iteration of the main loop (lines 10–19), a border point r is extracted from the open list and border points from the same quad considered for coverage. The extracted point indicates the cheapest point β_r it can reach. If β_r is already covered, then it is ignored because another

vertex already covered it with a cheaper path. If β_r is not yet covered, then it is covered by r. Once covered, β_r triggers a domain competition to determine its domain D_{β_r} in Q. As a consequence, the domain of other entry points on the same side may be updated. The details of these operations depend on the side of the quad. Next, we determine $\beta_{\beta_r} = p^*$, i.e., the cheapest reachable point from D_{β_r} and insert β_r into the queue with the cost to p^* as key. When the quad containing the target point is completely covered, the algorithm terminates and a fastest path from s to g can be constructed by backtracking the predecessor pointers from g to the starting point s, or more precisely, the fastest path from s to g is simply $\phi(g)$.

The function CompeteOnSide varies depending on sides. Before we describe it, we need to introduce some further elements.

Let $\mathcal{E}_Q^S$ be the set of entry points on side $S \in \{B, T, L, R\}$ in quad Q currently in $\mathcal{H}$ and let $\mathcal{E}_Q^{S\text{-}S'} \subseteq \mathcal{E}_Q^S$ be the subset of entry points on side S still active towards side S'. In order to retrieve entry points to the left and to the right of a point $p \in S$ efficiently, we maintain each $\mathcal{E}_Q^{S\text{-}S'}$ in the form of a balanced binary search tree (a red-black tree).

Let $p \in A_Q^S$ be a border point on some side $S \in \{B, T, L, R\}$ in a quad $Q \in \mathcal{Q}$. For two entry points $a, b \in \mathcal{E}_Q^B$, we say that a *dominates* b over p and we write $a \succ_p b$, if $t_a + t_{ap} \leq t_b + t_{bp}$ (ties may be broken arbitrarily). Likewise, if for *all* border points $p \in A_Q^S$ on a side $S \in \{B, T, L, R\}$, we have that $a \succ_p b$, we say that a *completely dominates* b on side S and write $a \succ_S b$. Alternatively, if there exists a border point p on side S for which $a \succ_p b$ we say that a *partially dominates* b on side S.

Compete on arrival side. For an entry point p on side B, the domain on the same side consists of the two closest points, one on the left and one on the right, including the first border point on the continuation of the side into the neighboring quads, if they exists. If both those points are themselves entry points or covered, then p is blocked on the arrival side. The cheapest reachable point will be one of those (at most two) points in the domain.

Compete on the opposite side (T). In domain competition on the opposite side T, we decide the entire domain D_a^T for each $a \in \mathcal{E}_Q^{B\text{-}T}$ when a new point b entering on the side B of the quad Q. The point b is determined by the pointer β_r of the point r extracted from the open list and it becomes an entering point after it has been covered. When an entry point a is the first entry point on side B of a new quad, its domain D_a^T will be assigned A_Q^T. However, when another entry point b arrives, the two entry points *compete* for their domains on side T. This means that A_Q^T is partitioned into two subsets $D_a^T \subseteq A_Q^T$ and $D_b^T \subseteq A_Q^T$.

Letting each entry point compete for its domain with every other entry point would lead to a worst case time complexity of $\mathcal{O}(n^2 \log(Kn))$ for competing on side T ($\mathcal{O}(\log(Kn))$ is the extraction cost of an entry point from $\mathcal{H}$). Instead, we do this in $\mathcal{O}(n \log(Kn))$ time by extending an algorithm from [6] to the case of vector weights. In that article, costs are determined by Euclidean distances. This allows to easily determine domination between points and save competitions. For example, for a quad $Q \in \mathcal{Q}$, an entry point a at $(x_a, 0)_Q$ dominates another entry point b at $(x_b, 0)_Q$, if $x_a < x_b$ and $t_a < t_b$. This condition does not hold trivially with vector weights. However, we can derive something similar with a bit more work.

For a new entry point b arriving on B, we need to compete with the active entry points in $\mathcal{E}_Q^{B\text{-}T}$ to its left and right in order of increasing distance.

For a point a to the left (right) of b, the following cases may arise:

1. a completely dominates b on T: the examination on the left (right) can be prematurely terminated since all further points in $\mathcal{E}_Q^{B\text{-}T}$ on the left (right) of a would also dominate b.

2. b completely dominates a on T: a becomes blocked on side T and the examination continues with the next point in $\mathcal{E}_Q^{B\text{-}T}$ to the left (right) of a.
3. b partially dominates a on T: the domains of b and a must be updated, but the examination to the left (right) can be prematurely terminated because no further points in $\mathcal{E}_Q^{B\text{-}T}$ to the left (right) of a will need to update its domain (since the domain of a will not be empty and its influence on other points from $\mathcal{E}_Q^{B\text{-}T}$ different from b was already determined).

When examining points from $\mathcal{E}_Q^{B\text{-}T}$, we update $\underline{i}_b$ if moving to the left and $\bar{i}_b$ if moving to the right. The update occurs only in Cases 2 and 3. After this update, if b is not blocked on T, it becomes part of $\mathcal{E}_Q^{B\text{-}L}$.

▶ **Theorem 3.1** (Separation point). *For two points $a, b \in A_Q^B$, $Q \in \mathcal{Q}$, domination and domain computation on side T can be carried out in constant time by determining the point $z = (x_z, \lambda_L)_Q$ that separates the domains. The value x_z is found by solving*

$$t_b + t_{bz}(\vec{w}_Q) = t_a + t_{az}(\vec{w}_Q). \tag{2}$$

For points a to the left of b, if $x_z > \bar{i}_a$, then we are in Case 1, while if $x_z < \underline{i}_a$, then we are in Case 2. Otherwise we are in Case 3 and x_z is the new value of $\bar{i}_a$ and $\underline{i}_b$. Similar derivations can be made for points a to the right of b.

Although x_z can be determined in constant time, the required retrieval and storage involves searching and updating the heap. This can be done in time amortized $\mathcal{O}(\log(Kn))$ per point. Thus, the total cost for carrying out domination for one side of a quad is $\mathcal{O}(n \log(Kn))$.

Once the domains are updated, we need to determine the cheapest reachable point from D_b^T.

▶ **Theorem 3.2** (Cheapest reachable point). *For an entry point p in A_Q^B, $Q \in \mathcal{Q}$, the point $z^* \in D_p^T$ that minimizes the travel time can be found by solving the following optimization problem:*

$$z^* = \arg\min\{f(z) = t_{p,z}(\vec{w}_Q) \mid z = (x_z, \lambda_T), x_z \in \mathbb{R}\}.$$

The problem has the closed form solution $x_{z^*} = \frac{w_1 \lambda_T}{w_2 - h}$, where $\vec{w}_Q = (w_1, w_2)_Q$.

Compete on the lateral sides (L and R). Competition on the two lateral sides is conducted in a manner very similar to the opposite side. The general algorithm with the three cases remains the same and separation points and cheapest points can be derived in constant time. Consequently, also the overall running time of $\mathcal{O}(n \log(Kn))$ is preserved.

Note that a further small speed up is possible on the opposite and lateral sides. For a first arriving entry point p at side B of a quad Q, let z^* be the cheapest reachable point on a side $S \in \{T, L, R\}$ and let $t_{\min}$ be $t_{pz^*}(\vec{w}_Q)$. Let also z_0 and z_{k_S-1} be the first and last border point of side S, respectively, and $t_{\max} = \max\{t_{pz_0}(\vec{w}_Q), t_{pz_{k_S-1}}(\vec{w}_Q)\}$. Then all later arriving entry points q will be blocked if $t_q > t_p + t_{\max} - t_{\min}$.

This is because p will be able to cover all points on the side S before q could cover them. This would already be discovered later by realizing that p completely dominates q (Case 1), but the above observation could save some cumbersome calculations. With regard to the runtime, the following holds:

▶ **Theorem 3.3** (Overall Time and Space Complexity). *A cheapest path in a vector-weighted tessellation $\mathcal{Q}$ of size K of a polygon F with n border points on each side of the quads of $\mathcal{Q}$ can be found in $\mathcal{O}(Kn \log(Kn))$ time using $\mathcal{O}(Kn)$ space.*

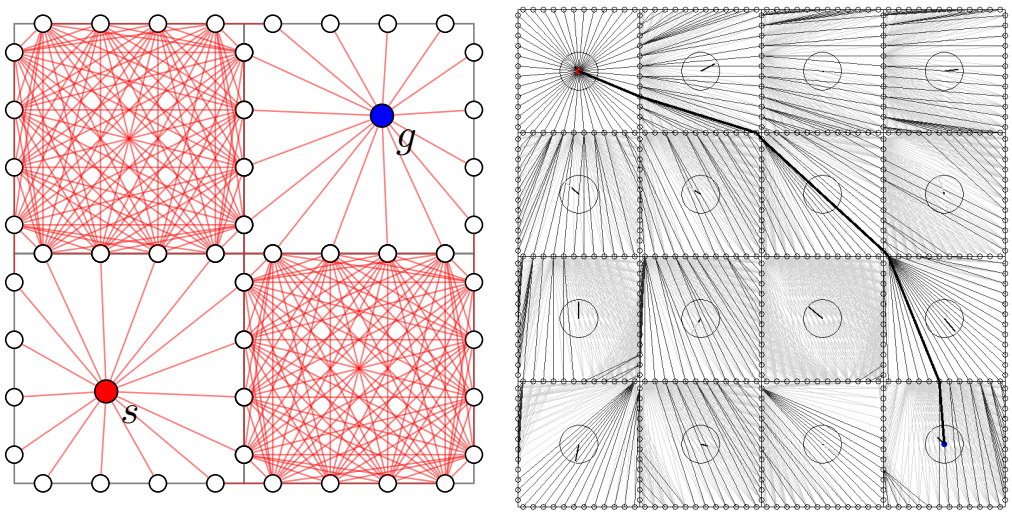

Figure 3 Left: four quads with four border points on each side and the source s and goal g nodes together with the corresponding graph G_{sg} with its vertices (circles) and edges (red segments). Right: The execution of our geometric algorithm on a 4×4 tessellation with source in the top-left quad and goal in the bottom-right quad. The path in bold black represents the optimal path, which deviates considerably from the straight line. The black lines represent the work done by our algorithm and the gray lines the work done by Dijkstra.

4 Experimental Analysis

For our experiments, we use rectangular polygons with tessellations consisting of $2i \times 3i$ square quads, $i = 2, \ldots, 10$. We also consider different choices for the number of border points $n \in \{9, 12, 18, 21, 25, 42\}$ on each side of the quads. So we consider instances of the problem with up to $51,891$ border points. For varying n, the inter-point distance is kept as constant as possible. Thus, the polygon grows in size as n increases. The wind vectors associated with each quad are generated independently at random as follows: Let ω be the magnitude and θ be the angle of the vector. We set $\vec{w} = (\omega, \theta) = (r_1 \cdot 2/3 \cdot h, r_2 \cdot 2\pi)$, where r_1, r_2 are sampled independently and uniformly at random from $[0, 1)$ and $h = 50$ is the speed of the aircraft.[6] In all instances, we consider a query for a path from a source s located in the center of the top left quad to a goal g located in the center of the bottom right quad.

In Fig. 3 right, we give evidence that cheapest paths may deviate considerably from the straight line path. The figure shows a screenshot of the execution of our geometric algorithm in a tessellation $\mathcal{Q}$. The circle in each quad has radius $\hat{h} = 2/3 \cdot h$ and indicates the maximum intensity of the wind, while the segment from the center indicates the actual wind vector. The bold black path indicates the cheapest path. The black segments are part of the cheapest tree from s to each border point in $\mathcal{Q}$. As expected, quads with strong headwind are avoided by cheapest paths. The gray segments belong to those paths that Dijkstra relaxed during its search when a better path to a node is found and they represent, therefore, the work saved by our geometric algorithm.

[6] We omit the unit of measure which is irrelevant for synthetic data.

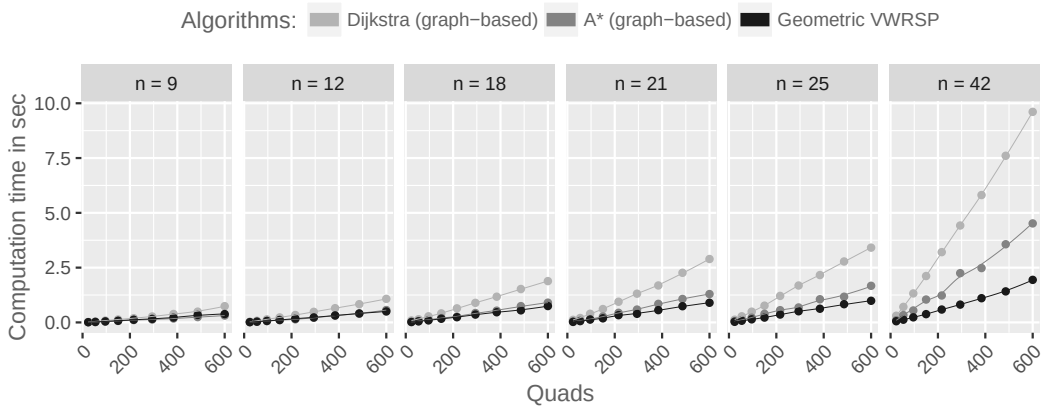

Figure 4 Comparison of running time between our algorithm (geometric vector-weighted region shortest path) and the two graph-based ones. The best run-times out of 5 runs are reported and a local regression line is superimposed on the points. The number of border points n on a side of each quad is increased in the panels from left to right.

Scaling. We study how the algorithms described (Dijkstra's, $A^\star$ search, and our geometric algorithm) scale with respect to increasing: size of the (rectangular) tessellations and number of border points. At each tessellation size and number of border points, we create three instances as explained above. For each instance and algorithm, we perform 5 runs and record the best running time.

Results are shown in Fig. 4. We see that for instances of the same number of quads, the running time increases with n. Dijkstra's algorithm consistently performs worst for any number of border points. $A^\star$ performs slightly better than our algorithm for $n = 9$, but as we increase the number of border points, our algorithm performs better than $A^\star$ search, with a margin that increases with n.

In practical situations, where the size of the polygon is fixed, increasing the number of border points allows us to decrease the inter-point distance among them and therefore naturally we want as many border points as possible, as this increase the accuracy of solutions. Thus, it is relevant to opt for our algorithm with the best running time performance.

Solution quality. We argued that the accuracy of the solutions increases when increasing the number of border points. In Fig. 5, we show two cases where this statement becomes evident. For an instance with 16×24 quads, we show the variation of solutions attained by our algorithms when increasing the number of border points from 1 to 25 with intervals of 1, while keeping the wind data constant. In particular, the solution represented in black is attained with the highest resolution ($n = 25$) while the others are depicted in gray. We observe that in these two cases, substantial differences in the paths arise. However, in general, the differences among the final cost of the solutions were less impressive. We argue that this behavior is due to the specific data that we used. That is, the instances were randomly generated and the quantities were dimensionless. While the number of quads 16×24 is realistic – for example, the case of FRA between Denmark and Sweden has 12×20 quads – the sizes of the quads and the inter-point distances were probably not. Hence, we believe that our analysis does not provide conclusive answers in this regard. A more precise assessment of the variation of solution quality with respect to resolution must necessarily include real-life data and quantify the variations in monetary terms.

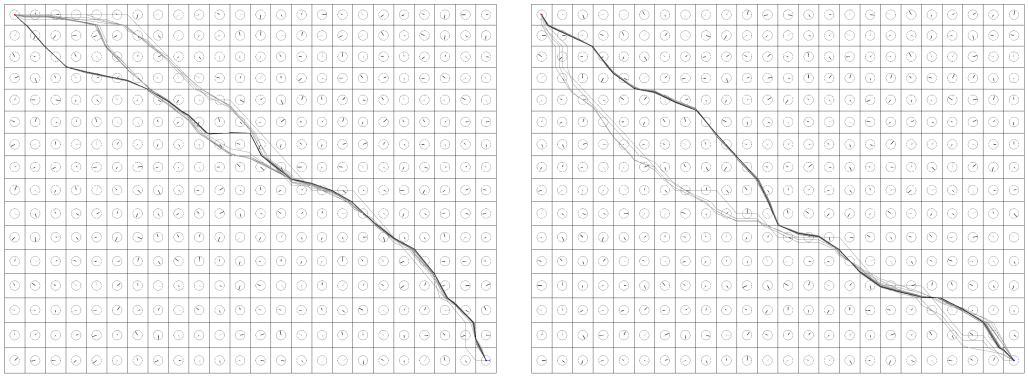

Figure 5 Visual comparison of solutions attained by our algorithm for increasing number of border points. For an instance with 16×24 quads, we increased the number of border points from 1 to 25 while keeping the wind data constant. In black, we show the solution attained with the highest resolution ($n = 25$) while the others are shown in gray.

5 Conclusions and Future Work

We have proposed a geometric algorithm for finding a shortest path in a polygon with a tessellation (partitioning) in vector-weighted regions. The algorithm has practical relevance in the context of finding flight routes in free route airspaces. Our algorithm is able to find routes that exploit wind forecast and because of this it is more effective than what is currently used in practice. We have provided a theoretical analysis of the running time of our algorithm, and experimentally, we have compared the running time of our algorithm with other classic graph-based algorithms such as Dijkstra and $A^\star$. Thanks to our set-up, our geometric algorithm is able to find the same routes as these algorithms, but its running time grows much more slowly when the tessellation size and the number of border points increase.

When computation time is an issue, it is possible to approximate optimal paths by joining adjacent regions of similar vector-weights. We have designed two heuristic algorithms for joining regions and studied the impact of these operations on the quality and on the computational cost of our algorithms. Although not reported here for reasons of space, this further analysis has shown another favorable feature of our algorithm. While joining quads and keeping the overall number of border points constant yields an increase in running time for classic Dijkstra and $A^\star$ because the number of edges increases, our algorithm is able in fact to decrease its running time. This result indicates that our algorithm offers more flexibility with respect to computation time than the other two.

We have used travel time as the cost to be minimized. In practice, the monetary cost rather than travel time is used, which depends on fuel consumption as well. The viability of our approach in that context has to be understood better. Further, we could not find an easy way to extend our geometric considerations to $A^\star$ search algorithms. Investigations in this direction would be interesting for future research. Finally, the flight route optimization problem is in fact a 3D problem that we have simplified to a 2D problem. Extensions of our ideas to three dimensions would be interesting.

References

1 Lyudmil Aleksandrov, Anil Maheshwari, and Jörg-Rüdiger Sack. Approximation algorithms for geometric shortest path problems. In *32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 286–295, 2000.

2 Steve Altus. Effective flight plans can help airlines economize. *AERO*, 35, 2009.

3 Marco Blanco, Ralf Borndörfer, Nam-Dung Hoang, Anton Kaier, Adam Schienle, Thomas Schlechte, and Swen Schlobach. Solving Time Dependent Shortest Path Problems on Airway Networks Using Super-Optimal Wind. In *16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2016)*, volume 54 of *OpenAccess Series in Informatics (OASIcs)*, pages 12:1–12:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.ATMOS.2016.12.

4 John Canny and John Reif. New lower bound techniques for robot motion planning problems. In *28th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 49–60, 1987.

5 Danny Z. Chen. Efficient algorithms for geometric shortest path query problems. In *Handbook of Combinatorial Optimization*, pages 1125–1154. Springer, 2013.

6 Danny Z. Chen, Robert J. Szczerba, and John J. Uhran. A framed-quadtree approach for determining Euclidean shortest paths in a 2-D environment. *IEEE Transactions on Robotics and Automation*, 13(5):668–681, 1997.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

8 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

9 Eurocontrol. Free route airspace (FRA). http://www.eurocontrol.int/articles/free-route-airspace. Accessed: 2017-03-28.

10 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

11 Andrew Freedman. Planes flew from New York to London at near-supersonic speeds due to powerhouse jet stream. *Mashable*, 9 Jan 2015. http://mashable.com/2015/01/08/jet-stream-new-york-london-flights/ Accessed: 2017-05-30.

12 Jongrae Kim and João Hespanha. Discrete approximations to continuous shortest-path: Application to minimum-risk path planning for groups of UAVs. In *42nd IEEE Conference on Decision and Control (CDC)*, pages 1734–1740, 2003.

13 Anders N. Knudsen, Marco Chiarandini, and Kim S. Larsen. Constraint handling in flight planning. In *23nd International Conference on Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science. Springer, 2017. To appear.

14 Anders Nicolai Knudsen, Marco Chiarandini, and Kim S. Larsen. Vertical optimization of resource dependent flight paths. In *22nd European Conference on Artificial Intelligence (ECAI)*, pages 639–645, 2016. doi:10.3233/978-1-61499-672-9-639.

15 Joseph S. B. Mitchell and Christos H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of the ACM*, 38(1):18–73, 1991.

16 Robert J. Szczerba, Danny Z. Chen, and John J. Uhran. Planning shortest paths among 2D and 3D weighted regions using framed-subspaces. *The International Journal of Robotics Research*, 17(5):531–546, 1998.

17 WAFC Washington. The world area forecast system (WAFS) internet file service (WIFS) users guide. https://www.aviationweather.gov/wifs/docs/WIFS_Users_Guide_v4.1.pdf. Accessed: 2017-05-30.