Midterm Assignment 2

March 16, 2021

Indicaciones generales

1. Fecha de publicación: 16 de marzo de 2021.
2. Fecha de entrega: 24 de marzo de 2020 hasta las 23:55.
3. Único medio de entrega: `https://e-aulas.urosario.edu.co`.
4. Formato de entrega: código en Python.
5. Importante: no use acentos ni deje espacios en los nombres de los archivos que cree.
6. La actividad **debe** realizarse **individualmente**.
7. Los grupos pueden consultar sus ideas con los profesores para recibir orientación; sin embargo, la solución y detalles del ejercicio debe realizarlos **individualmente**. Cualquier tipo de fraude o plagio es causa de anulación directa de la evaluación y correspondiente proceso disciplinario.
8. El grupo de trabajo debe indicar en su entrega de la solución a la actividad cualquier asistencia que haya recibido.
9. El grupo no debe consultar ninguna solución a la actividad que no sea la suya.
10. El grupo no debe intentar ocultar ningún código que no sea propio en la solución a la actividad (a excepción del que se encuentra en las plantillas).
11. Las entregas están sujetas a herramientas automatizadas de detección de plagio en códigos.
12. `e-aulas` se cerrará a la hora acordada para el final de la evaluación. La solución de la actividad debe ser subida antes de esta hora. El material entregado a través de `e-aulas` será calificado tal como está. Si ningún tipo de material es entregado por este medio, la nota de la evaluación será 0.0.

Introduction

An essential part in aviation is the planning of flights. Given a departure and a destination, a flight plan is a carefully designed route to take safely an aircraft between those two points, within an estimated time en route. As expected, not every flight might be pleasant so a flight plan must include contingency procedures, in case an emergency ensues. For that, several pieces of information are included: estimated time en route, alternate airports in case of detouring, type of flight, crew's information, number of passengers on board, and aircraft specifications. Flight planning is extremely important, especially when flying over unexplored areas, as they provide vital information in case of mechanical failure, bad weather, or when rescuing operations are needed.

Also known as flight path, a route consists of line segments located on a map that aid the aircraft to navigate from the departure to the destination airports. Although there are several types of routes, with different attributes such as altitude and airspeed, we will consider *a route to be defined as a sequence of two-dimensional line segments* on a map. A single flight plan may include more than one flight path; that is, different routes for different scenarios.

Flight planning also involves the identification of airports which can be flown to in case of unexpected conditions at the destination airport or along the route. The flight plan should only include alternate airports or locations which can be reached with the current fuel load and that have the capabilities necessary to handle the type of aircraft being flown.

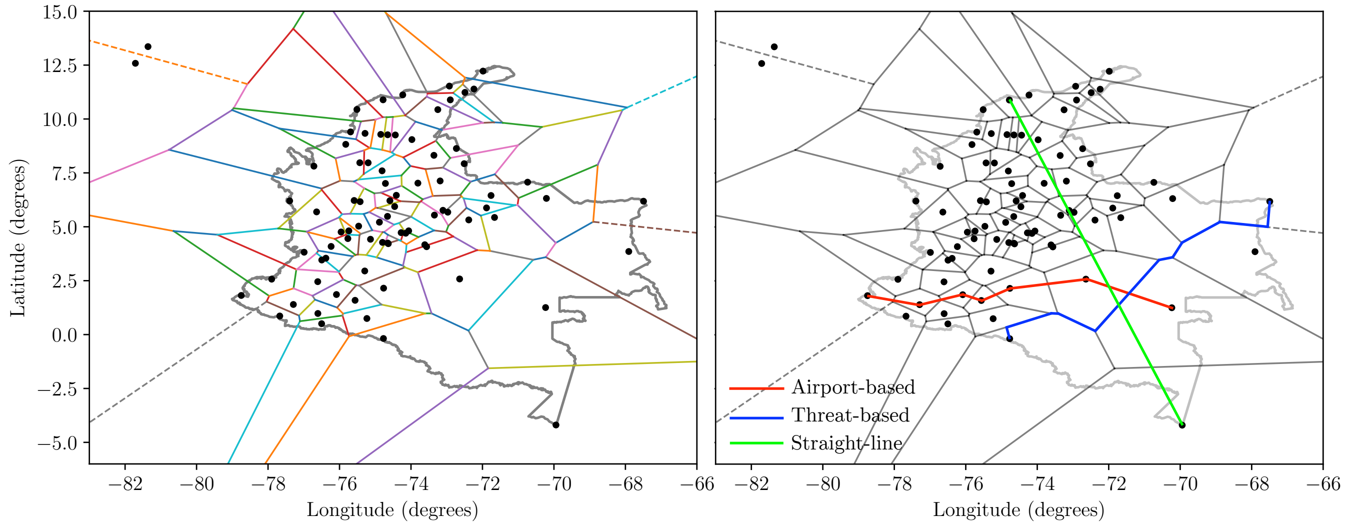

Figure 1: Left: Colombia's two-dimensional Voronoi diagram for flight space with airports as sites. Right: Examples of the several types of flight paths that might be calculated by the application.

Problem

Your task is to design and implement a computational application that addresses features of a flight plan. The functionality required evaluates aspects of the flight plan, given a set of airport sites, and departure and destination points of the flight. These aspects include global and local properties such as preprocessing of airport sites, nearest and farthest airports, types of flight paths or routes, potential threats, and desolated areas.

The application must be **implemented as a class with a collection of methods and its corresponding test suite**, coded in `Python 3`. We will treat locations or points on the map as defined by an ordered pair —latitude and longitude— and measure all distances using the standard Euclidean norm in the plane. The methods must satisfy the following specifications:

1. Read airport and border locations from input data files (enclosed with this document). The airport's input file contains extra information on altitude, cities and name. You may want to keep such information. This method must generate a plot similar to that of Fig. 1, left.
2. Airport coverage area. It is always important to know how large and what population is serviced by a given airport; this helps minimizing the ration of fuel to passenger number. Find the airports that cover the biggest and smallest areas of service. Notice that this can only be computed for airports with finite area of service.
3. Most and least crowded airports. In case of an emergency landing it is important to know how populated are the airports in order to cause the less damage. Report the names, not locations, of the airports with the smallest and the largest number of neighboring airports. Make sure that this search problem is solved in linear time in the structural complexity of the Voronoi diagram.
4. Airport-based flight path. Given two locations: departure and destination, construct a flight path as a constrained shortest path between these locations. The flight path is a collection of connected straight lines whose endpoints are airports; hence, it is *not* a straight line. Visiting airports along the flight, although expensive in gas, reduces the risk of landing on open fields

away from runways. The flight path must be represented as a list of locations (Fig. 1, right).

5. Threat-based flight path. In this case, the flight path must avoid airports in order to reduce potential collisions with other aircrafts, at the cost of not having a nearby runway to perform emergency landings. This flight path consists of a collection of connected line segments, with the beginning and end points corresponding to departure and destination locations. Again, the path is not a straight line and must cruise *equidistantly* from all neighboring airports. The flight path must be represented as a list of locations (see Fig. 1, right).

The class implemented should be named `FlightOperations`. You have the freedom to choose the attributes of the class; they should be chosen such that redundant computation is avoided as much as possible. Extra methods can be implemented if deemed necessary.

Strategy

Let us now analyze how to proceed. The input to the problem is the locations of the airports and the border map, so you need to read files and appropriately keep the information. Next, remembering what you learned in Computational Geometry, you know that you need to build a computational model of the network of airports so that you can efficiently process the information. A good model candidate is the Voronoi diagram, which is a computational model that subdivides the plane in cells. Each cell has a unique site (airport) assigned to it and consists of all points of the plane closer to that site than to any other in the plane (map).

Indeed, the Voronoi diagram's particular structure offers us the capabilities to answer the questions posed above, most of them optimally. These questions may be related to the following well-known problems in computational geometry:

⋆ *Nearest-neighbor search problem*. Given a set of points (airports) in the plane and a query point (location), find the point in the set that is closest to the given query point.

⋆ *Largest empty circle problem*. Given a set of points (airports) in the plane, find a largest circle centered within their convex hull and enclosing none of them.

⋆ *Closest pair of points problem*. Given a set of points (airports) in the plane, find a pair of points with the smallest distance between them.

⋆ *Shortest path problem*. This is the problem of finding a path between two vertices in a graph (airports or Voronoi vertices), always visiting an edge and without creating edges.

If the size of the set is n, these problems can be solved in $O(n \lg n)$ time. Understanding these problems and the algorithms that solve them is key to successfully completing your assignment.

We now have a clear picture (*hopefully!*) of how to proceed with the construction of the class.

References

F. Aurenhammer and R. Klein. Voronoi Diagrams. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 5, pages 201-290. North Holland, 2000.

M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Third edition. Springer-Verlag, 2008.