DESIGN OF ALGORITHMS USING PYTHON 2024/25

Maritime Border Delimiter

Parts 1 and 2 - Deadline for Checkpoint - 27 October

1 Introduction

The United Nations' Convention on the Law of the Sea (UNCLS) establishes a legal framework for the definition of each countries' continental shelf, making it possible to establish control over the seabed for territory that goes beyond the territorial waters and the Exclusive Economic Zone. In this context, Portugal (in parallel with many other countries) is negotiating these limits with the UN, working on a proposal for the delimitation of Portugal's seabed. The goal is to maximize its surface, while conforming to the rules established by the UNCLS. Asserting control over seabed territory entitles a country to both the exploration and protection of its resources. The aim of this project is to develop a tool that automatically reaches a proposal that fulfills these goals.

The rules establish natural and political criteria for justifying a valid delimitation of the seabed, and include both *permissive* and *restrictive* conditions. These criteria are complex (`https://en.emepc.pt/projeto-pepc-os-criterios`), but for the purpose of the project can be simplified in the following way:

1. A valid delimitation consists of a polygon line that has as vertices valid points, and such that the distance between each connected vertice cannot exceed a certain number of nautical miles (nmi);

2. Vertices can be chosen from a set of points on the seabed, based on the the thickness of its sediments, or the change in its slope.

These points result from analysis of measurements and geological samples collected by a Remote Operated Vehicle (ROV) that performs deep sea dives[1]. The negotiation process has been

[1]This research is performed by EMEPC (`https://en.emepc.pt/`), using the Portuguese ROV LUSO (`https://en.emepc.pt/rov-luso`).

going since Portugal's initial proposal in 2017, by means of meetings during which valid points are agreed upon, and new delimitation proposals must be recalculated, on-the-fly.

The tool to be developed should then aid in efficiently finding a delimitation with the largest area possible, while respecting the above criteria, so as to assist the negotiations with UNCLS. While it is meant to be generic, and work for any part of the world, the tool may assume that the data refers to the area of interest for Portugal, which allows to make some simplifying assumptions, such as that the delimited area will consist of a single polygon. Sections 3 and 3 specify the behaviour of the tool in more detail, including the format of the input and output that the tool should produce.

The project is an opportunity for you to practice programming in Python and to apply your knowledge on some data structures and algorithms learned in the course. The construction of the program is divided into two phases: the first focuses on defining useful abstract data-types that enable to reason more easily about the more advanced components of the program that are to be developed at the final phase.

1.1 Input format

Your program should handle data about points that is read from a file that is written in csv format. In each line it should contain, in this order, an identifier (a unique string), the latitude and the longitude of a point, separated by tabs. For example:

```
p1 0 10
p2 10 10
p3 3 15
p4 4 5
p5 4 12
p6 1 20
p7 7 3
p8 12 -1
```

2 Specification of the tool – Part 1

2.1 Commands

The aim of this first stage is to set up the basic abstract data types that your program will operate with, and to build the basic interaction cycle for providing inputs, and executing commands.

The tool should consist of a **single** `.py` file, containing all the code, and be callable in the command line, with no arguments:

```
$ python ./mytool.py
```

It should then enter a cycle where it waits for a new command and executes it, until a command for exiting the program is given. For now, the possible commands are the following:

These commands should be read from the standard input, and the result printed to the standard output, according to the following behavior.

import_points <file_name>
make_map
quit

Tabela 1: Commands for running the tool and their arguments.

`import_points <file_name>`**.** Reads data about points from `<file_name>`. The file `file_name` should have the format described in Section 1.1. The tool should import into Python all data in the file.

`make_map`**.** Creates a map with all the points, and the latest determined delimitation, if existing. The map should be stored as a new .html file, with a fresh name that depends on the instant in which it was created, with at least an accuracy of milliseconds. You can use Python's `time` module for this purpose. Example: "map_1727425700.387014.html"

`quit`**.** Exits the program.

You can assume that the types of the arguments are correctly introduced by the user (so you do not need to validate the types of the arguments). However, to help the debugging process, it is recommended that, if there can be an error in the process, the program should raise an exception of an appropriate type with an informative error message.

2.2 Abstract Data Types

Your tool should include and use an implementation of the abstract data types FileHandler, ValidPoints, Point and Delimitation, with the interfaces defined below.

1. To make sure that your program respects the abstraction barriers of the ADTs, **all field attributes of these classes should be hidden**, i.e., should be named using the double underscore: `__field_name`.

2. Document all of the classes and methods in order to make explicit: the **internal representation** of the objects of each class, as well as the **parameter and return types** of each method. Additionally, include the **complexity** of every function and method (see subsection 3.4).

FileHandler. The FileHandler ADT collects the program components that interact with files, including reading a specified file with data about points, in the format specified in Section 1.1, and producing html maps. It includes the following methods:

- `__init__ : FileHandler` $\times$ `str` $\rightarrow$

  Receives a new FileHandler object `self` and a file name as a string, and initializes the file handler (it is automatically returned at instantiation time).

- `read_points : FileHandler → ValidPoints`

  Receives the FileHandler object `self`, uses the file handler's file name to read data about points, creates the ValidPoints object, and uses it to store the Point objects corresponding to the data it imports from the file. It returns the ValidPoints object.

- `make_map : FileHandler × Delimitation ∪ {None} →`

  Receives the FileHandler object `self` and a Delimitation object representing a line or None, and creates an html file containing a map. The map should represent the points that have been stored in the file handler's ValidPoints object as circles, and the delimitation as a line. You can use Python's `folium` module to produce the map.

- At least the standard Python method `__repr__`, exhibiting the file name and the points that have been imported.

ValidPoints. The ValidPoints ADT is used to represent the collection of points that have been imported from a file, all of which are considered valid for use in the delimitation process. It offers the functionality for retrieving the imported information. It includes the following methods:

- `__init__ : ValidPoints × list(Point) × int ∪ {None} →`

  Receives a new ValidPoints object `self` and a list of Point objects, and initializes it (it is automatically returned at instantiation time).

- `get_size : ValidPoints → int`

  Receive the ValidPoints object `self`, and returns the number of points that are stored.

- `get_min_lat, get_max_lat, get_min_lon, get_max_lon : ValidPoints → float`

  These four different methods receive the ValidPoints object `self`, and return the minimum latitude, maximum latitude, minimum longitude and maximum longitude (respectively) of the points that are stored, as a float.

- `get_all_points : ValidPoints → list(Point)`

  Receives the ValidPoints object `self`, and returns the list of Point objects that are stored, sorted according to the identifier field.

- `get_points_vicinity : ValidPoints × Point × int → set(Point)`

  Receives the ValidPoints object `self`, a Point object, and a maximum distance as an integer, and returns the subset (the type is set) of Point objects that are stored and which are within the maximum distance **(in nautical miles)** of the given point.

- At least the standard Python method `__repr__`, exhibiting the points that are stored.

Point. The point ADT represents the data associated to a single Point.

- `__init__ : Point × str × float × float →`

  Receives a new Point object `self`, a string representing its identifier, its latitude as a float and its longitude as a float, and initializes it (it is automatically returned at instantiation time).

- `get_id : Point → str`

  Receives the Point object `self`, and returns the string representing its identifier.

- `get_latitude, get_longitude : Point → float`

  These two methods receives the Point object `self`, and return the float corresponding to its latitude or longitude (respectively).

- `distance : Point × Point → float`

  Receives the Point object `self` and a second Point object, and returns the distance between the two points as a float, **in nauical miles**. In order to calculate correct distances as measured on the planetary surface, i.e. *geodesic* distances, you can use the `distance` function of Python's `geopy distance` module (it uses geodesic distances by default).

- `in_vicinity_of : Point × Point × int → bool`

  Receives the Point object `self` a second Point object and an integer representing a maximum distance **(in nautical miles)**, and returns True or False depending on whether `self` is within the specified distance (using the `distance` operation above) from the second point.

- `get_forward_angle : Point × Point × Point →` **`float`**

  Receives the Point object `self`, a second Point object and a third Point object, and returns the angle **(in degrees)** that is formed between the semi-line that runs from the first point and through the second point, and the one running from the first point and the third point. **The angle should be measured using a *sinusoidal* projection of the Earth on a plane.**

- At least the standard Python methods `__repr__` and `__eq__` (representing strict equality of point identifiers, latitudes and longitudes).

Delimitation. The Delimitation ADT represents a complete or incomplete polygon, connecting points on the globe. When polygons are to be considered as complete (determined by the context in which they are used), there is an implicit connection between the first and the last points. The ADT should support the drawing of the polygon, where points that are added consecutively are considered to be connected. It provides operations that are sensitive to the order in which the points are added.

- `__init__ : Delimitation →`

  Receives a new Delimitation object `self`, and initializes it (it is automatically returned at instantiation time).

- `get_points : Delimitation → list(Point)`

  Receives the Delimitation object `self`, and returns the list of Point objects that form the delimitation. The list should respect the order in which points were added to the delimitation, such that any pair of consecutive points in the list are connected in the polygon. For complete polygons, also the first and last are considered connected.

- `get_first : Delimitation → Point`

  Receives the Delimitation object `self`, and returns the first point that was added to it.

- `get_last_two : Delimitation → Point × Point`

  Receives the Delimitation object `self`, and returns a tuple containing the last two Point objects that have been added to the delimitation (the rightmost one should be the last one added).

- `get_area : Delimitation → float`

  Receives the Delimitation object `self`, and returns a float corresponding to the area of the polygon that is represented, **in square nautical miles. The area should be measured using a *sinusoidal* projection of the Earth on a plane.**

- `size : Delimitation → int`

  Receives the Delimitation object `self`, and returns the number of points that it connects as an integer.

- `add_point : Delimitation × Point →`

  Receives the Delimitation object `self` and a Point object, and adds the given point to the delimitation.

- `pop_point : Delimitation → Point`

  Receives the Delimitation object `self`, and removes and returns the last Point object that had been added to the delimitation.

- `copy : Delimitation → Delimitation`

  Receives the Delimitation object `self`, and returns a new Delimitation object that is a copy of the given one, and contains Point objects that are copies of the original delimitation's.

- `intersects : Delimitation × Point × Point × Point × Point → bool`

  Receives a Delimitation object `self` and four Point objects, and determines whether the segment formed by the first two intersects the segment formed by the latter two, returning a boolean value accordingly.

- `crosses_delimitation : Delimitation × Point × Point → bool`

  Receives the Delimitation object `self` and two Point objects, and determines whether the segment formed by the two points intersects the delimitation, returning a boolean value accordingly. It is necessary that if the new segment ends in the first point of the delimitation, while not intersecting any other segment belonging to the delimitation, that this is *not* considered to be a crossing.

- `show : Delimitation × ValidPoints →`

  Receives the Delimitation object `self` and a ValidPoints object, and displays a window where all the stored points are displayed along with their identifier, and the complete delimitation formed by lines in between (possibly a subset of the) points is plotted. You can use Python's `matplotlib.pyplot` module.

- At least the standard Python method `__eq__`, that should return True if the Delimitation objects that are being compared represent the same complete polygon (note that there are several ways of delimiting the same polygon), and `__repr__`, exhibiting the list of points that form the delimitation.

2.3 Analysis

For each method and function, **include in the docstring its complexity using the big-O notation**. It should be expressed in terms of the number of points (P).

2.4 Preparing your project for evaluation

To enable automatic evaluation of your project, please consider the following:

- Note that names, format, argument types and number of commands, basic operations and classes, as well as the types and all details related to the output of the functions, need to be *exactly* as specified, in order to avoid failing the automatic tests.

- Besides the Python modules mentioned in this project specification, you can use math, copy and pythonds3. Do not use any other module without checking if it is allowed.

- Make sure that you insert any "main" blocks of code under a test.

3 Specification of the tool – Part 2

In this final stage of the project, you should develop, test and analyse algorithms for finding delimitations, given a set of valid points.

3.1 Notes on delimitations

When *no* maximum distance is imposed between points in a delimitation, the optimal delimitation coincides with the *convex hull* (`https://en.wikipedia.org/wiki/Convex_hull`), i.e., a polygon that envelopes all of the points being considered, and which by definition is an entirely convex shape. A convex hull may be visualized as the shape enclosed by a rubber band stretched around a finite set of points.

However, when a maximum distance *is* imposed between points in a delimitation, the optimal delimitation might not be entirely convex, i.e., a convex hull that respects those restrictions might not exist.

Finding an optimal delimitation for a set of points is a difficult problem, and it might be necessary to resort to a brute force approach. However, when time efficiency is of essence, an approximate solution might be useful nevertheless. The tool that is being developed in this project should be able to determine delimitations corresponding to three main scenarios:

1. No maximum distance is imposed between points in a delimitation;
2. A maximum distance is imposed between points in a delimitation, and area optimality should be prioritized;
3. A maximum distance is imposed between points in a delimitation, and time efficiency should be prioritized;

3.2 Commands

The tool should consist of a **single** `.py` file, containing all the code, and be callable in the command line, with no arguments:

```
$ python ./mytool.py
```

It should then enter a cycle where it waits for a new command and executes it, until a command for exiting the program is given. The possible commands are the following:

These commands should be read from the standard input, and the result printed to the standard output, according to the following behavior.

`import_points <file_name>`**.** As before, reads data about points from `<file_name>`. The file `file_name` should have the format described in Section 1.1. The tool should import into Python all data in the file. **Additionally, it should print the time used to import the points.**

`import_points <file_name>`
`draw_hull`
`draw_optimal_delimitation <max_distance>`
`draw_approx_delimitation <max_distance>`
`make_map`
`quit`

Tabela 2: Commands for running the tool and their arguments.

`draw_hull`**.** Determines the hull that wraps all of the valid points that have been imported. Prints the complete delimitation, its area, and the time used to calculate it.

`draw_optimal_delimitation <max_distance>`**.** Determines a delimitation using only valid points, that respects the maximum distance that is given as argument, and which is optimal, i.e., has the maxmim area that is possible. Prints the complete delimitation, its area, and the time used to calculate it.

`draw_approx_delimitation <max_distance>`**.** Determines a delimitation using only valid points, that respects the maximum distance that is given as argument, and which efficiently approximates as much as possible the optimal delimitation. Prints the complete delimitation, its area, and the time used to calculate it.

`make_map`**.** As before, creates a map with all the points, and the latest determined delimitation, if existing. The map should be stored as a new .html file, with a fresh name that depends on the instant in which it was created, with at least an accuracy of milliseconds. You can use Python's `time` module for this purpose. Example: "map_1727425700.387014.html" **Additionally, it should print the name of the html file that was produced, and the time used to produce the map.**

`quit`**.** Exits the program, **printing a closing message**.

You can assume that the types of the arguments are correctly introduced by the user (so you do not need to validate the types of the arguments). However, to help the debugging process, it is recommended that, if there can be an error during execution, the program should raise an exception of an appropriate type with an informative error message.

Output format. The format of the output produced by the commands `import_points` should be exactly the following

```
Size:<one space><number>
Time:<one space><time>
<newline>
```

where space><number> is the number of points that were imported from the file, as an integer; and <time> is in milliseconds and appears as a float with 2 decimal positions.

The format of the output produced by the commands draw_* should be exactly the following

```
Type:<one space><type>
Line:<one space><delimitation>
Area:<one space><area>
Time:<one space><time>
<newline>
```

where type can be one of three strings – Convex Hull or Optimal Delimitation or Approximate Delimitation –; <delimitation> is printed according to the specification of __str__ for Point and for Delimitation; <area> is in square nautical miles, as a float with 2 decimal positions; and <time> is in milliseconds and appears as a float with 2 decimal positions.

The format of the output produced by the commands make_map should be exactly the following

```
Map:<one space><file name>
Time:<one space><time>
<newline>
```

where <file name> is the name of the html file containing the map; and <time> is in milliseconds and appears as a float with 2 decimal positions.

The format of the output produced by the commands quit should be exactly the following

```
Done!
<newline>
```

Note: All times should be calculated using Python's time or timeit modules.

3.3 Abstract Data Types

Your tool should include and use an implementation of the abstract data types FileHandler, ValidPoints, Point and Delimitation, with the interfaces as defined in Part 1 of the project. Additionally, it should implement the additional methods for these ADTs, as well as the new ADTs ConvexHull, ApproxDelimitation and OptimalDelimitation, according to the interface defined below. You can of course define other auxiliary ADTs. As mentioned above, you can use the pythonds3 module. **If you need to make any changes to the code in pythonds3, it should be included in your solution file.**

1. To make sure that your program respects the abstraction barriers of the ADTs, **all field attributes of these classes should be hidden**, i.e., should be named using the double underscore: __field_name.

2. Document all of the classes and methods in order to make explicit: the **internal representation** of the objects of each class, as well as the **parameter and return types** of each method. Additionally, include the **complexity** of every function and method (see subsection 3.4).

Point. The Point ADT should implement an additional operation for printing a point in a particular format:

- `__str__ : Point → str`

  Receives a new Point object `self`, and returns a string representing the point, with precisely the following format

  `(<id>, <latitude>, <longitude>)`

  where `<latitude>` and `<longitude>` are written as floats with **the same precision as the imported data** ~~two decimal positions~~.

Delimitation. The Delimitation ADT should implement additional operations for recognizing a valid delimitation, and for printing a delimitation in a particular format:

- `is_valid_delimitation : Delimitation × ValidPoints × int → bool`

  Receives a new Delimitation object `self`, a ValidPoints object, and an integer representing a maximum distance in nautical miles, and returns True or False depending on whether `self` is a valid delimitation, i.e.:

  - all points in the delimitation belong to the given set of valid points;
  - distances between consecutive points of the **complete** delimitation do not exceed the given maximum distance;
  - it does not cross itself.

- `__str__ : Delimitation → str`

  Receives a new Delimitation object `self`, and returns a string representing the delimitation, with precisely the following format

  `[<Point 1>, <Point 2>,... , <Point n>]`

  where `n` is the number of points in the delimitation, `<Point i>` are printed using the Point's `__str__`, in the order in which they were added into the delimitation.

ConvexHull. The ConvexHull ADT determines the delimitation that is a convex hull corresponding to a set of valid points. It includes the following methods:

- `__init__ : ConvexHull × ValidPoints →`

  Receives a new ConvexHull object `self` and a ValidPoints object, and initializes the drawing of the hull (it is automatically returned at instantiation time).

- `find_delimitation : ConvexHull → Delimitation`

  Receives the ConvexHull object `self`, calculates the Delimitation object defining the convex hull, and returns it.

OptimalDelimitation. The OptimalDelimitation ADT determines an optimal delimitation corresponding a set of valid points. It includes the following methods:

- `__init__ : OptimalDelimitation × ValidPoints × int →`

  Receives a new OptimalDelimitation object `self`, **a ValidPoints object** and an integer representing the maximum distance between points in a delimitation, and initializes the drawing of the delimitation (it is automatically returned at instantiation time).

- `find_delimitation : OptimalDelimitation → Delimitation`

  Receives the OptimalDelimitation object `self`, and returns a Delimitation object that represents an optimal delimitation.

ApproxDelimitation. The ApproxDelimitation ADT determines an approximation of the optimal delimitation corresponding to a set of valid points. It includes the following methods:

- `__init__ : × ApproxDelimitation × ValidPoints × int →`

  Receives a new ApproxDelimitation object `self`, **a ValidPoints object** and an integer representing the maximum distance between points in a delimitation, and initializes the drawing of the delimitation (it is automatically returned at instantiation time).

- `find_delimitation : ApproxDelimitation → Delimitation`

  Receives the ApproxDelimitation object `self`, calculates a Delimitation object that approximates the optimal delimitation, and returns it.

3.4 Analysis

For each method and function, **include in the docstring its complexity using the big-O notation**. It should be expressed in terms of the number of points (P).

3.5 Preparing your project for evaluation

To enable automatic evaluation of your project, please consider the following:

- Note that names, format, argument types and number of commands, basic operations and classes, as well as the types and all details related to the output of the functions, need to be *exactly* as specified, in order to avoid failing the automatic tests.
- Besides the Python modules mentioned in this project specification, you can use math, copy and pythonds3. Do not use any other module without checking if it is allowed.
- Make sure that you insert any "main" blocks of code under a test.

4 Development, submission and evaluation

Important Dates. The development of the Project will be structured as follows:

Part 1: The interaction cycle, basic ADTs for reading, storing and analysing data. **Deadline 13 October**.

Full Submission: Additionally to the requirements of Part 1, it shall implement algorithms for finding optimal (or approximately optimal) delimitations. **Deadline 27 October** ~~November~~.

Practical test and discussion: About the project developed by each group, but performed **individually**. Week **28 October to 31 October**, though it may be necessary to postpone discussions due to time incompatibilities.

Testing. Examples are provided to assist in testing your code. It is however your responsibility to perform more extensive testing to ensure the correctness and robustness of the tool. For this purpose, you can create small examples that help you check specific details of the program.

Support. During both theoretical and practical classes you will be offered guidance for the development of the project. Tips will be provided in the course's webpage and Zulip platform. If you need any help, do not hesitate to ask. You are encouraged to use the Zulip stream `#project` for questions and open discussions about the project.

Authorship ad plagiarism. Projects are to be solved in groups of 2 elements. Both members of the group are expected to be equally involved in solving, writing and understanding the project, and share full responsibility for all aspects of all components of the evaluation. You can exchange ides with your colleagues, but each group is expected to write all of its own code. Presence at the practical test and discussion with tool demonstration is mandatory in order to have a grade.

Any form of plagiarism consisting of submitting portions of code that has not been written by the group, be it from people or by AI, with or without the consent of those who wrote it, will entail failing the project and reporting to the School. This applies to both the code receivers and the code providers. Have in mind that facilitating plagiarism is not helping, and cheating does not lead to learning. If you need extra help, please do seek to contact the lecturer.

Submission and evaluation. The project should be submitted for automatic evaluation according to instructions that will be made available in the course's website.

The weight of each phase of the project is 50% each, and will be evaluated according to:

1. The group's submission (75%), including:

   (a) How well it meets the specification;

   (b) The efficiency of the implementation;

   (c) The analysis and programming style (worth up to 20% of the submission's grade);

   The baseline grade for the group's submission will be obtained using automatic and manual evaluation according to the above criteria.

2. The student's practical test and discussion (25%). During the practical test and discussion, each student in the group is expected to be able to demonstrate knowledge of all details of the submitted code in order to be graded for the project. If failing to do so, the student's final project grade can be changed to reflect the demonstrated knowledge.

Good work!