Analysis on current solution for ra2ce project.

Done on 25/11/2021 By Carles S. Soriano Pérez (carles.sorianoperez@deltares.nl)

Project structure

1. Cleanup of unnecessary files from root directory.

    1. Makefile purpose.
        1. Seems this might be used for documentation (SPHINX).
        2. Please allocate in its proper directory (docs or else).
    2. Test files should be inside tests/test-data (or similar).
    3. Architectural / Flow aiagrams are better placed in docs/diagrams (or similar).
    4. Should be moved:
        1. Dirs: settings, data/folder_structure_template.
        2. Files: analysies.ini, ra2ce overview.drawio, run.py, test_config.json.

2. Update imports. Currently it's a bit mixed up what needs to be installed from conda (wheels and platform dependencies) and what from requireemnts / setup (in fact the latter does not point to any requirements file).

    1. Suggestion. Migrate to poetry, it brings great results and can be easily adapted by github pipelines. It will require rewriting the setup as it should be moved to the project.toml standard.

3. Although a linter is defined (flake8), there is no action to prevent it from failing.

    1. I have added an action in ci.yml which will format the latest pushed code with black (pep8) to prevent this.
    2. I have also added an updated requirements.txt file and updated the environment.yml file to make the above pipeline work (though it will fail during the test run).

Licensing

Licensing is subject to the solution owners purposes. Hereby we discern a few cases:

1. Solution can be used and code freely modified for their own interest.
2. Solution can be used, code modified, but cannot make commercial use of it.
3. Solution can only be used as a library, applications using it can be commercially distributed.
4. Solution can only be used as an application, modifying it source its permitted as long as remains with the same level

In Deltares we usually base our solutions in the following Licenses:

- MIT - https://opensource.org/licenses/MIT
- LGPL - https://opensource.org/licenses/lgpl-license
- GPL - https://opensource.org/licenses/gpl-license

If we try to map the licenses tot he previous foreseen cases, we could do it such as:

1. MIT, when we do not constrain interested parties from doing as they want with the solution (and the code).
2. LGPL, when we require interested parties to also distribute their software under the same conditions as we do.
3. LGPL, if they use the solution just a library (by importing it or just a .dll).
4. GPL.

There are many other licenses for open source code available. However it is recommended to stay within the line of work of Deltares so that, at least, we are acknowledge over the propietrarity of the original code.

Code style

1. Pep8 is suggested, yet not enforced (mentioned in previous section). Suggestion to use a code formatter such as black and also isort for ordering imports at the beginning of each file.

2. Docstrings. There are indeed docstrings in almost all methods. However their style differ.

    1. Suggestion is to uniform them and enforce them during code review.
    2. Personal advice on google format (check this for more reference Google PyGuide)
    3. It is worth going through all the files and updating / adding docstrings where needed.

3. Type hinting is a must (and helps autodocstrings indeed enabled).

4. Personally I do not see the need to claim the author of a file specially when using git(hub). It becomes outdated pretty soon.

Architecture

- The biggest flaw at this moment is the lack of separation of concerns between classes and readers / writers.
- An expected workflow for this is such as:

```
[File] -> [Reader] -> [FileObjectModel] -> [Configure] -> [DataObjectModel]
```

```
1. A reader class for that particular file type.
2. An object (or data structure) representing the file being read (with all its content).
3. A separate class where a FOM is given and can be configured returning a DOM.
```

* When using external libraries, it is still good to wrap their usage in our own classes instead of just using their own.

```
[File] -> [ExternalLibrary] -> [ConfigureWrapper] -> [DataObjectModel]
```

- Files where this can be applied / seen:
    - origins_destinations.py: Mixed loging for reading multiple files and configuring an individual one.
        - Create a raster reader.
        - Create a ra2ce graph utils class / wrapper with methods to modify the graph:
            - create_od_pairs
            - find_closes_vertice
            - find_new_nearest_vertice
            - split_line_with_points
            - cut
            - getKeysByValue
            - add_od_nodes.
    - networks.py:
        - Suggestion to inverse network creation so it is done from a factory:
            - Given config -> identify type -> Generate Network type:
                - NetworkShp
                - NetworkOsmPbf
                - NetworkOsmDownload
            - Alternatively make the current methods into class methods and init the class giving a config.
    - analyses_direct.py / analyses_indirect.py:
        - There's duplication of code here (for instance save_gdf).
        - Advice to have one class per file.
        - Extract stand-alone methods into a separate utils class.
        - Create readers / writers accordingly (for instance the save_gdf).
        - Watch out with method naming.
        - EffectivenessMeasures:
            - Static methods should only be used
            - Static methods for reading -> external class.
        - Roaddamage:
            - All its methods could actually be static methods.
    - direct_lookup.py:
        - Another reader that could be extracted.
        - I believe this could better be addressed by defining them as classes and having the entries as objects as well.
            - The idea is to improve maintainability with the data structures.
        - Pydantic library can be of great help to get this.
    - multi / single _link.py:
        - This gives me the impression that it might be wise to start restructuring the code to apply correctly patterns and introduce protocols for better uniformity (specially if the code will be open for collaborations).

- Architectural patterns:
    - Ideally you would apply a factory and adapter pattern to the networks (depending on their origin they get built one way or another, yet the resulting object should be virtually the same).
    - Similarly, you also want to treat the analysis with a 'Façade' pattern. This means you have an upper layer with the classic init / update / run / finalize and then each different analysis implements it in its own way.
        - Note that you still need to implement the concrete direct and indirect ones as it's already done. This is mostly a formality to ensure them both belong to the same abstraction / concept (Python introduced Protocols which are quite cool to use combined with factories).

Quality assurance.

A first analysis is done by SonarQube (VPN required). SonarQube

By applying some architectural concepts we can better test all the intermediate steps both by means of unit and integration test.