extremitypathfinder

Release /bin/sh: 1: poetry: not found

Jannik Michelfeit
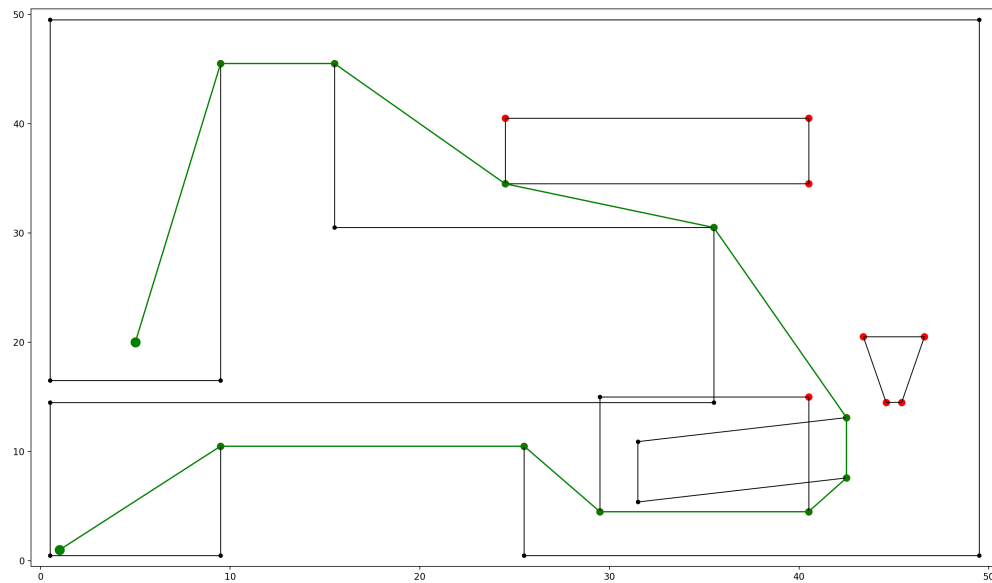
CONTENTS:

python package for geometric shortest path computation in 2D multi-polygon or grid environments based on visibility graphs.

GETTING STARTED

1.1 Installation

Installation with pip:

```
pip install extremitypathfinder
```

Installation with Numba for a significant speedup, with the tradeoff of a larger installation footprint and slight initial compilation time (until caching kicks in):

```
pip install extremitypathfinder[numba]
```

1.2 Dependencies

please refer to the `pyproject.toml` file for current dependency specification.

1.3 Basics

```python
from extremitypathfinder import PolygonEnvironment

environment = PolygonEnvironment()
# counter clockwise vertex numbering!
boundary_coordinates = [(0.0, 0.0), (10.0, 0.0), (9.0, 5.0), (10.0, 10.0), (0.0, 10.0)]
# clockwise numbering!
list_of_holes = [
    [
        (3.0, 7.0),
        (5.0, 9.0),
        (4.5, 7.0),
        (5.0, 4.0),
    ],
]
environment.store(boundary_coordinates, list_of_holes, validate=False)
start_coordinates = (4.5, 1.0)
goal_coordinates = (4.0, 8.5)
path, length = environment.find_shortest_path(start_coordinates, goal_coordinates)
```

All available features of this package are explained *HERE*.

USAGE

Note: Also check out the *API documentation* or the code.

2.1 Initialisation

Create a new instance of the *PolygonEnvironment class* to allow fast consequent timezone queries:

```python
from extremitypathfinder import PolygonEnvironment

environment = PolygonEnvironment()
```

2.2 Store environment

Required data format: Ensure that all the following conditions on the polygons are fulfilled:

- numpy or python array of coordinate tuples: `[(x1,y1), (x2,y2,)...]`
- the first point is NOT repeated at the end
- must at least contain 3 vertices
- no consequent vertices with identical coordinates in the polygons (same coordinates in general are allowed)
- a polygon must NOT have self intersections
- different polygons may intersect each other
- **edge numbering has to follow this convention (for easier computations):**
    - outer boundary polygon: counter clockwise
    - holes: clockwise

```python
# counter clockwise vertex numbering!
boundary_coordinates = [(0.0, 0.0), (10.0, 0.0), (9.0, 5.0), (10.0, 10.0), (0.0, 10.0)]

# clockwise numbering!
list_of_holes = [
    [
        (3.0, 7.0),
```

```
        (5.0, 9.0),
        (4.5, 7.0),
        (5.0, 4.0),
    ],
]
environment.store(boundary_coordinates, list_of_holes, validate=False)
```

Note: Pass `validate=True` in order to check the condition on the data. Raises `TypeError` if the input has the wrong type and `ValueError` if the input is invalid.

Note: If two Polygons have vertices with identical coordinates (this is allowed), paths through these vertices are theoretically possible! When the paths should be blocked, use a single polygon with multiple identical vertices instead (also allowed).
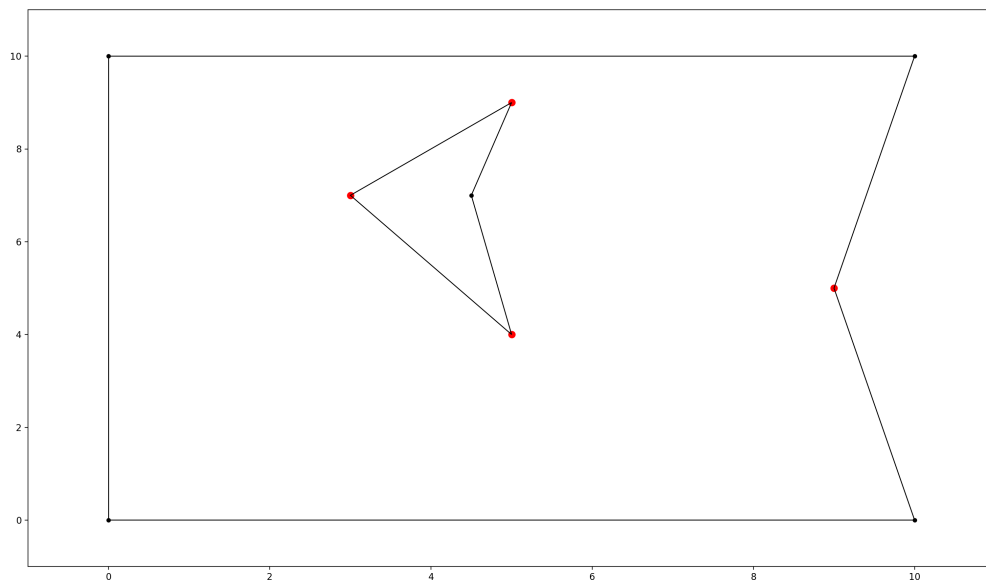
Fig. 1: polygon environment with extremities marked in red

2.3 Visibility Graph Pre-computation

Storing the map properties automatically computes the *visibility graph* of the environment once.
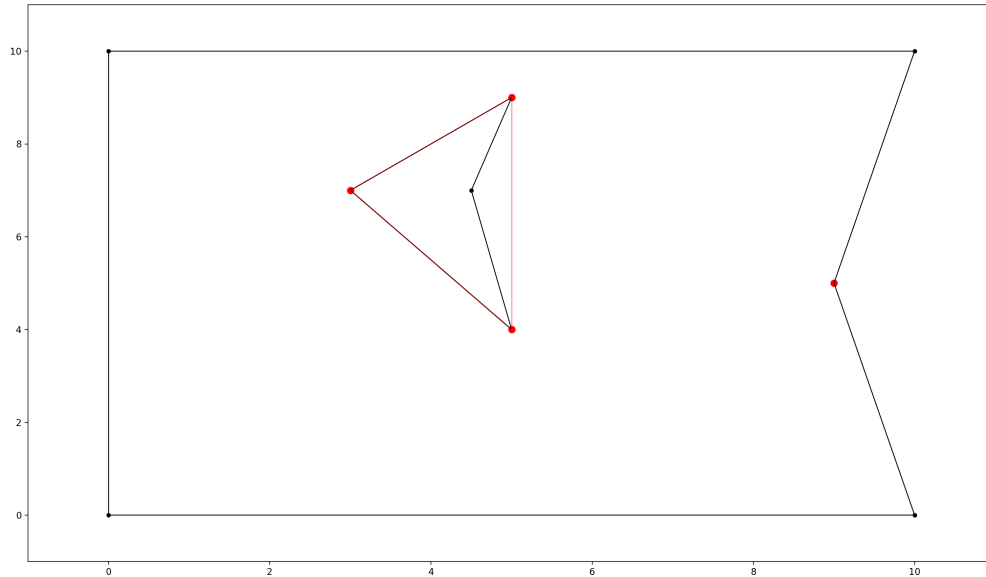
Fig. 2: polygon environment with optimised visibility graph overlay in red

2.4 Query

```
start_coordinates = (4.5, 1.0)
goal_coordinates = (4.0, 8.5)
path, length = environment.find_shortest_path(start_coordinates, goal_coordinates)
```

If any start and goal point should be accepted without checking if they lie within the map, set `verify=False`. This is required if points lie really close to polygon edges and "point in polygon" algorithms might return an unexpected result due to rounding errors.

```
path, length = environment.find_shortest_path(
    start_coordinates, goal_coordinates, verify=False
)
```
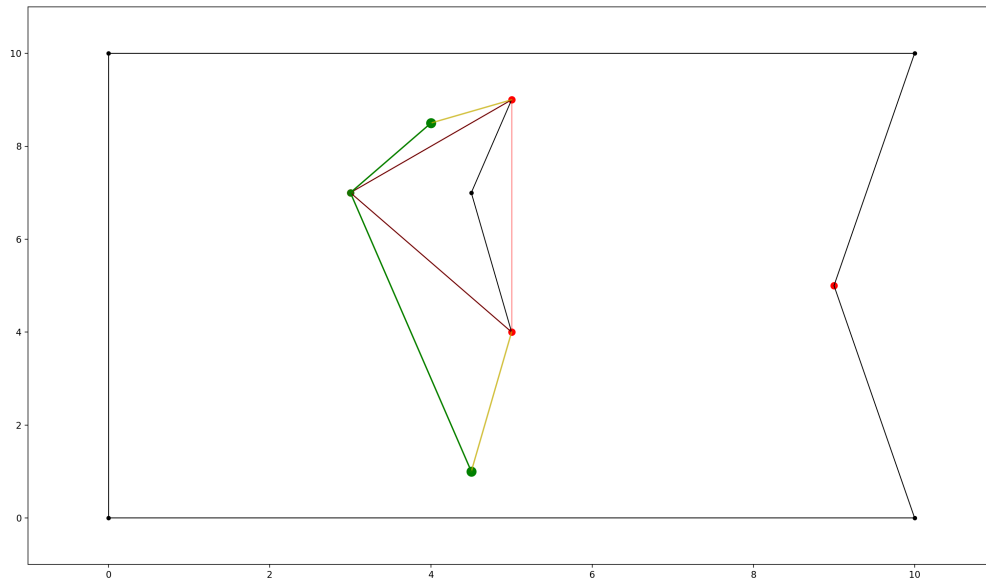
Fig. 3: polygon environment with optimised visibility graph overlay. visualised edges added to the visibility graph in yellow, found shortest path in green.

2.5 Converting and storing a grid world

```
size_x, size_y = 19, 10
obstacle_iter = [  # (x,y),
    # obstacles changing boundary
    (0, 1),
    (1, 1),
    (2, 1),
    (3, 1),
    (17, 9),
    (17, 8),
    (17, 7),
    (17, 5),
    (17, 4),
    (17, 3),
    (17, 2),
    (17, 1),
    (17, 0),
    # hole 1
    (5, 5),
    (5, 6),
    (6, 6),
    (6, 7),
    (7, 7),
    # hole 2
```

(continues on next page)

```
    (7, 5),
]
environment.store_grid_world(
    size_x, size_y, obstacle_iter, simplify=False, validate=False
)
```
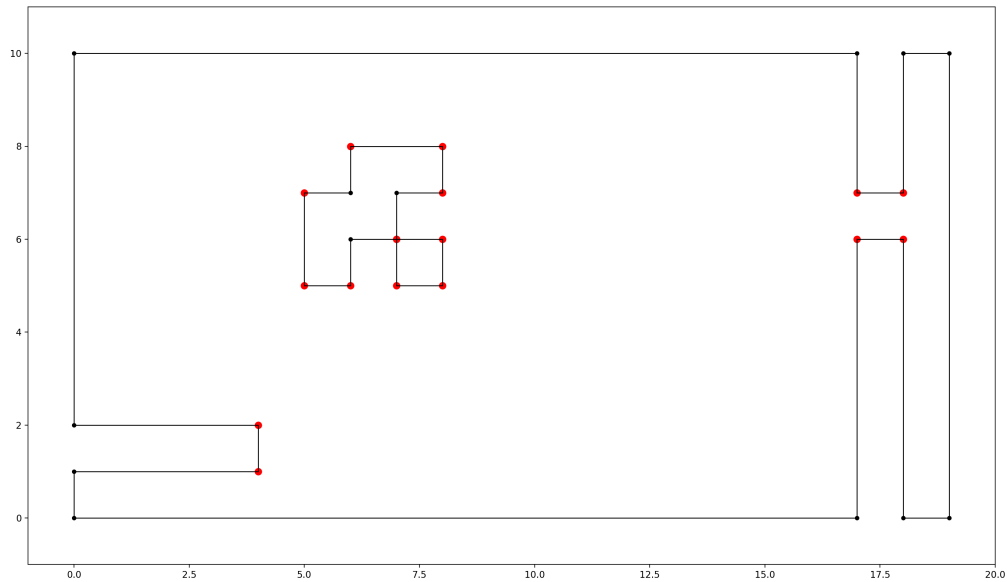

Fig. 4: grid-like environment converted to a polygon environment with "extremities" marked in red

Note: As mentioned in [1, Ch. III 6.3] in "chessboard-like grid worlds" (many small obstacles have a lot of extremities!) it can be better to use A* right away (implemented in `graph_search.py`).

2.6 Cache and import the environment

```
environment.export_pickle(path="./pickle_file.pickle")

from extremitypathfinder.extremitypathfinder import load_pickle

environment = load_pickle(path="./pickle_file.pickle")
```

2.7 Plotting

The class `PlottingEnvironment` automatically generates plots for every step in the path finding process:

```python
from extremitypathfinder.plotting import PlottingEnvironment

environment = PlottingEnvironment(plotting_dir="path/to/plots")
environment.store(boundary_coordinates, list_of_holes, validate=True)
path, distance = environment.find_shortest_path(start, end)
```

Other functions in `plotting.py` can be utilised to plot specific parts of an environment (extremities, edges, …)

2.8 Calling extremitypathfinder from the command line

A command line script is being installed as part of this package.

Command Line Syntax:

```
extremitypathfinder <path2json_file> -s <start> -g <goal>
```

The <start> and <goal> arguments must be passed as two separate float values.

Example:

```
extremitypathfinder ./example.json -s 2.5 3.2 -g 7.9 6.8
```
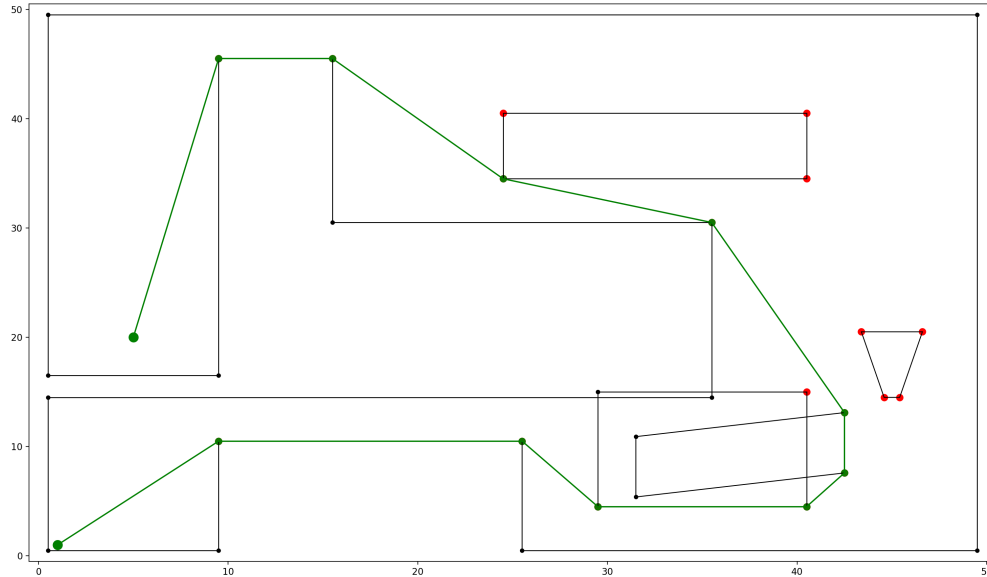
This returns [(2.5, 3.2), (5.0, 4.0), (7.9, 6.8)] 6.656009823830612

Please note that this might be significantly slower than using the package directly from within python.

ABOUT

python package for fast geometric shortest path computation in 2D multi-polygon or grid environments based on visibility graphs.

Also see: GitHub, PyPI

3.1 License

`extremitypathfinder` is distributed under the terms of the MIT license (see LICENSE).

3.2 Basic Idea

Well described in [1, Ch. II 3.2]:

An environment ("world", "map") of a given shortest path problem can be represented by one boundary polygon with holes (themselves polygons).

IDEA: Two categories of vertices/corners can be distinguished in these kind of environments:

- protruding corners (hereafter called **"Extremities"**)
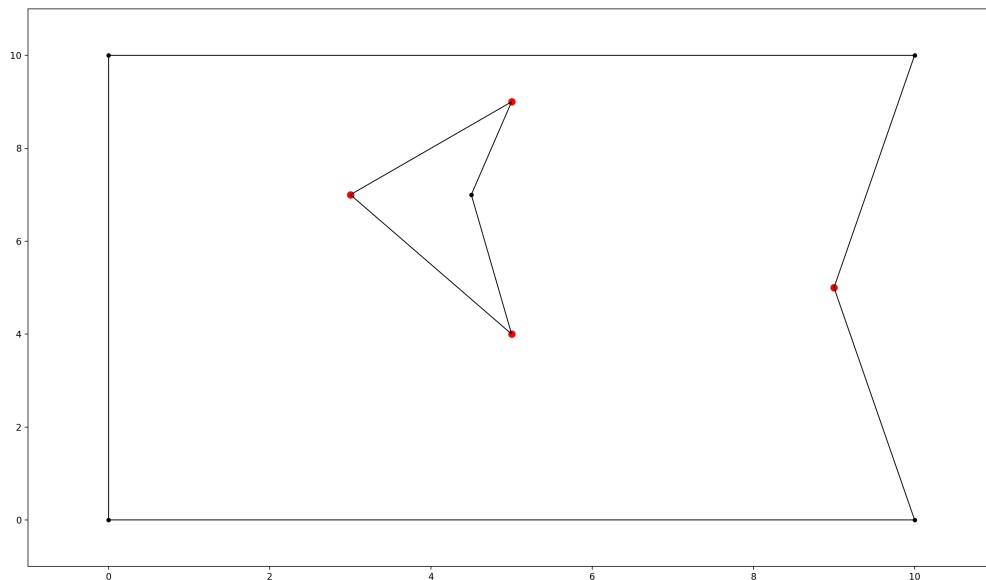
- all others

Fig. 1: polygon environment with extremities marked in red

Extremities have an inner angle (facing towards the inside of the environment) of > 180 degree. As long as there are no obstacles between two points present, it is obviously always best (=shortest) to move to the goal point directly. When obstacles obstruct the direct path (goal is not directly 'visible' from the start) however, extremities (and only extremities!) have to be visited to reach the areas "behind" them until the goal is directly visible.

Improvement: As described in [1, Ch. II 4.4.2 "Property One"] during preprocessing time the visibility graph can be reduced further without the loss of guaranteed optimality of the algorithm: Starting from any point lying "in front of" an extremity `e`, such that both adjacent edges are visible, one will never visit `e`, because everything is reachable on a shorter path without `e` (except `e` itself). An extremity `e1` lying in the area "in front of" extremity `e` hence is never the next vertex in a shortest path coming from `e`. And also in reverse: when coming from `e1` everything else than `e` itself can be reached faster without visiting `e1`. -> `e` and `e1` do not have to be connected in the graph.

3.2.1 Algorithm

This package pretty much implements the Visibility Graph Optimized (VGO) Algorithm described in [1, Ch. II 4.4.2], just with a few computational tweaks:

Rough Procedure:

- **1. Preprocessing the environment:** Independently of any query start and goal points the optimized visibility graph is being computed for the static environment once. Later versions might include a faster approach to compute visibility on the fly, for use cases where the environment is changing dynamically. The edges of the precomputed graph between the extremities are shown in red in the following plots. Notice that the extremity on the right is not connected to any other extremity due to the above mentioned optimisation:
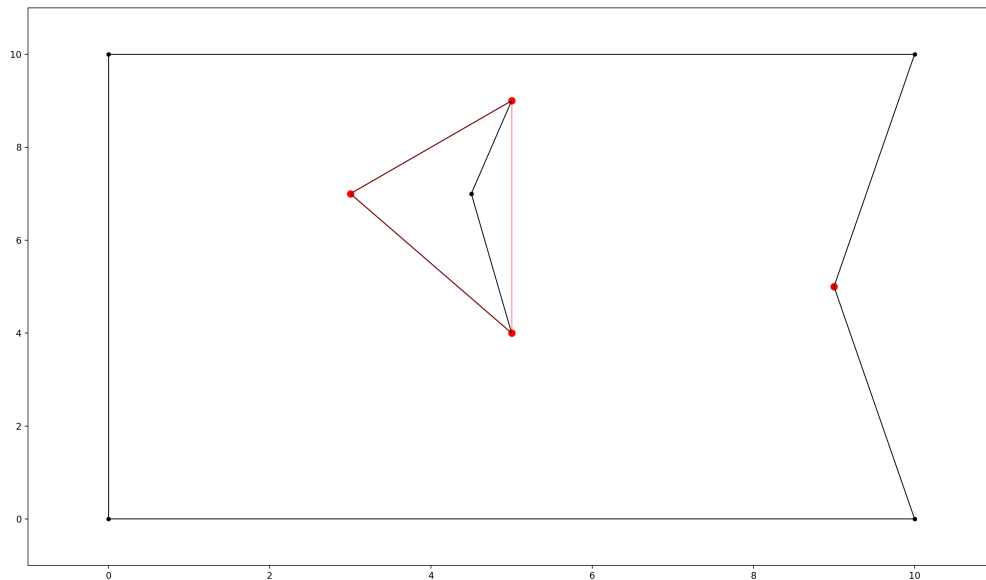
Fig. 2: polygon environment with optimised visibility graph overlay in red

- **2. Including start and goal:** For each shortest path query the start and goal points are being connected to the internal graph depending on their visibility. Notice that the added edges are directed and also here the optimisation is being used to reduce the amount of edges:

- **3. A-star shortest path computation :** Finding the shortest path on graphs is a standard computer science problem. This package uses a modified version of the popular `A*-Algorithm` optimized for this special use case.

Fig. 3: optimised directed heuristic graph for shortest path computation with added start and goal nodes

Fig. 4: polygon environment with optimised visibility graph overlay. visualised edges added to the visibility graph in yellow, found shortest path in green.

Edge Case: Overlapping Vertices and Edges

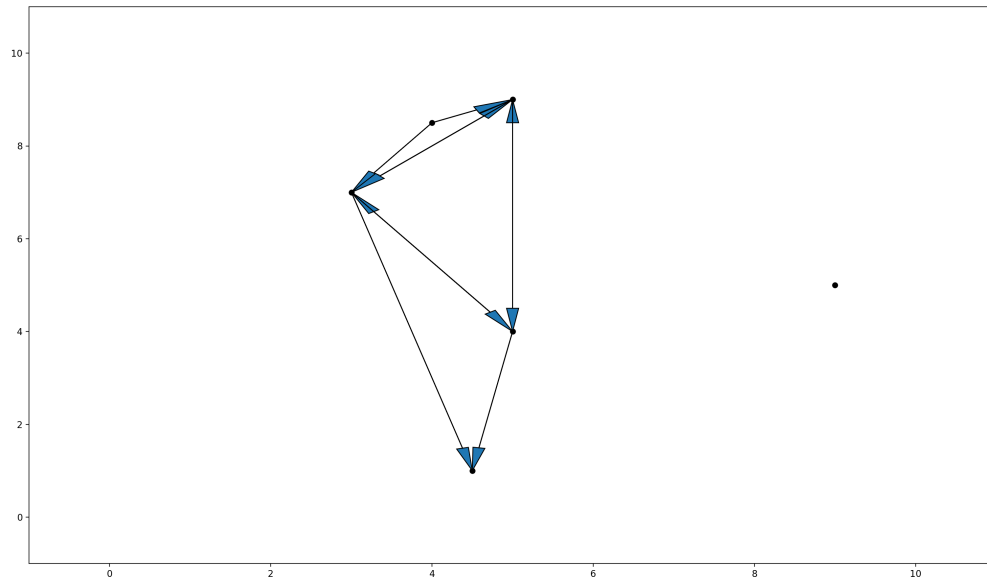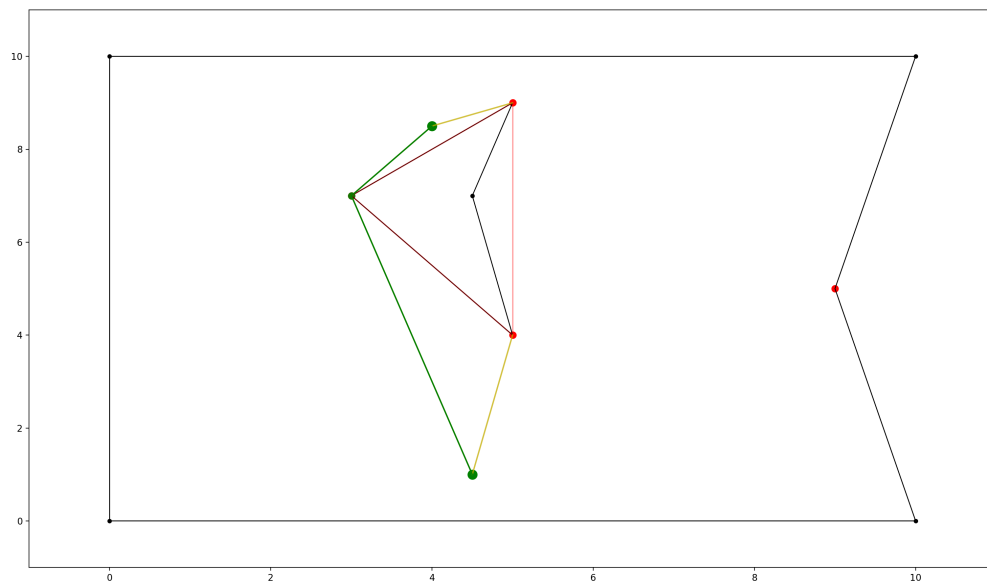> **Warning:** Overlapping edges and vertices are considered "non-blocking". Shortest paths can run through either. Ensure Holes and Boundary Polygons are truly intersecting and not just touching in order to "block" paths.

3.2.2 Implementation

AREA is an algorithm for computing the visibility graph.

In this use case we are not interested in the full visibility graph, but the visibility of just some points (all extremities, start and goal).

Simple fundamental idea: points (extremities) are visible when there is no edge running in front "blocking the view".

Rough procedure: For all edges delete the points lying behind them. Points that remain at the end are visible.
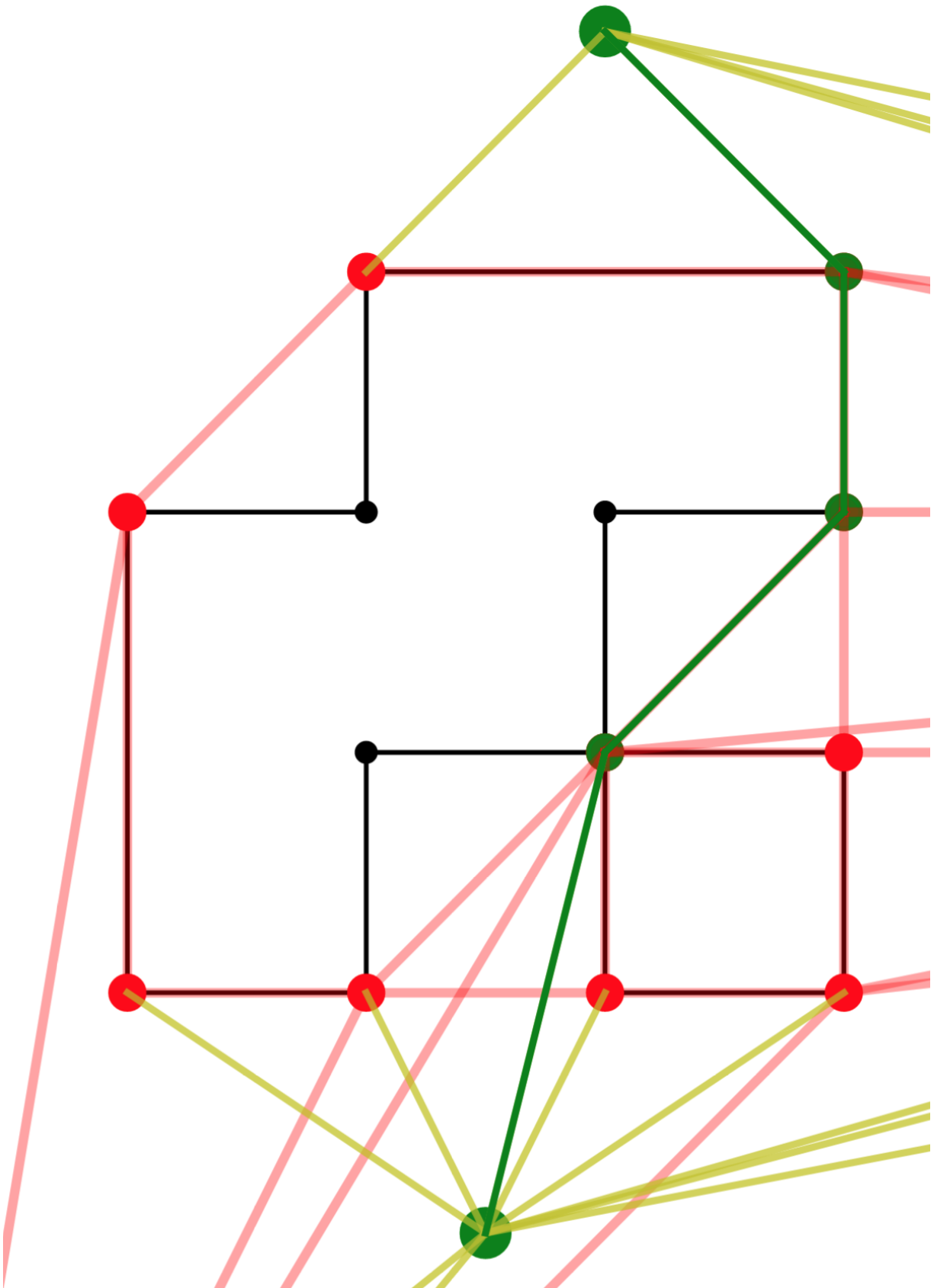
Optimisations:

- for each edge only checking the relevant candidates ("within the angle range"): * By sorting the edges after their angle representation (similar to Lee's algorith, s. below), only the candidates with a bigger representation have to be checked. * By also sorting the candidates, the candidates with a smaller representation than the edge don't have to be checked.

- angle representations: instead of computing with angles in degree or radians, it is much more efficient and still sufficient to use a representation that is mapping an angle to a range $a \in [0.0; 4.0[$ ($[0.0; 1.0[$ in all 4 quadrants). This can be done without computationally expensive trigonometric functions!

- deciding if a point lies behind an edge can often be done without computing intersections by just comparing distances. This can be used to reduce the needed computations.

Properties:

- checking all edges

- checking an edge at most once

- ability to process only a subset of all vertices as possible candidates

- decreasing number of candidates after every checked origin (visibility is a symmetric relation -> only need to check once for every candidate pair!)

- no expensive trigonometric computations

- actual Intersection computation (solving linear scalar equations) only for a fraction of candidates

- could theoretically also work with just lines (this package however currently just allows polygons)

Runtime Complexity:

- m: the amount of extremities (candidates)

- n: the amount of edges / vertices (since polynomial edges share vertices), with usually $m << n$

- $O(m)$ for checking every candidate as origin

- $O(nlog_2 n)$ for sorting the edges, done once for every origin -> $O(mnlog_2 n)$

- $O(n)$ for checking every edge -> $O(m(nlog_2 n + n)$

- $O(m/n)$ (average) for checking the visibility of target candidates. only the fraction relevant for each edge will be checked -> $O(m(nlog_2 n + (nm)/n) = O(m(nlog_2 n + m) = O(mnlog_2 n$

- since m n the final complexity is $O(mnlog_2 n) = O(n^2 log_2 n)$

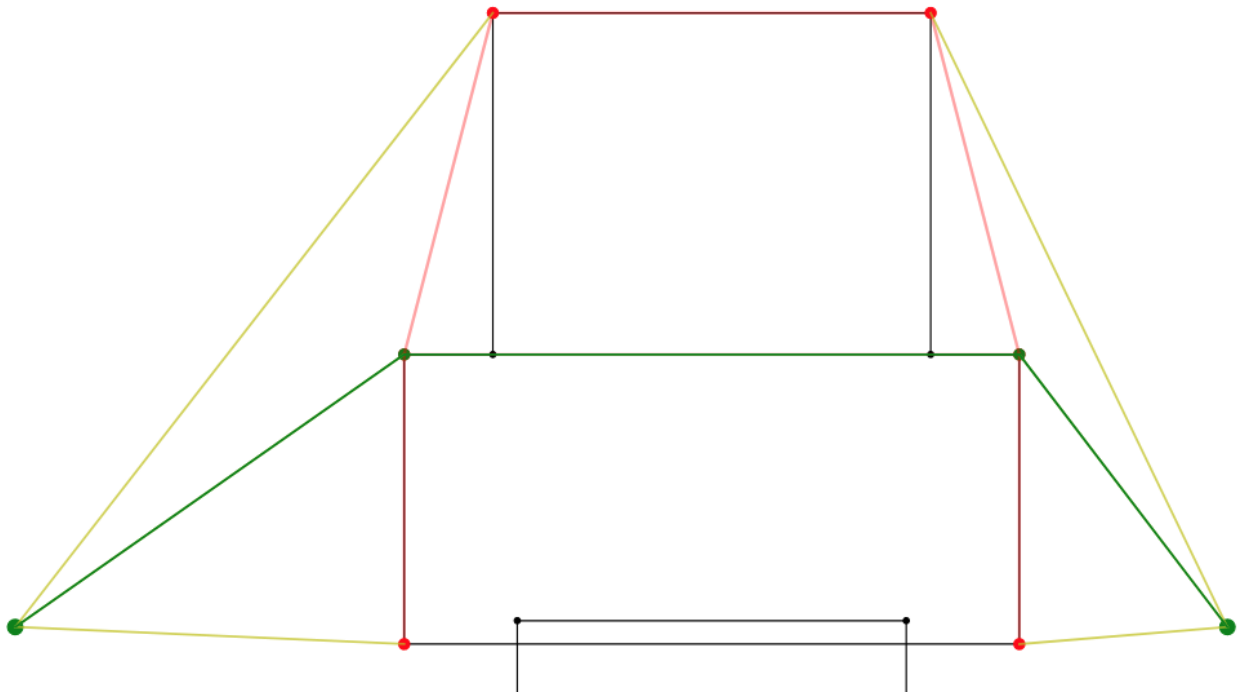Fig. 5: example of a shortest path running through overlapping vertices

Fig. 6: example of a shortest path running along two overlapping edges

The core visibility algorithm (for one origin) is implemented in `PolygonEnvironment.get_visible_idxs()` in `extremitypathfinder.py`

Lee's visibility graph algorithm:

complexity: $O(n^2 log_2 n)$ (cf. these slides)

- Initially all edges are being checked for intersection

- Necessarily checking the visibility of all points (instead of just some)

- Always checking all points in every run

- One intersection computation for most points (always when T is not empty)

- Sorting: all points according to degree on startup, edges in binary tree T

- Can work with just lines (not restricted to polygons)

Currently using the default implementation of A* from the *networkx* package.

Remark: This geometrical property of the specific task (the visibility graph) could be exploited in an optimised (e.g. A*) algorithm:

- **It is always shortest to directly reach a node instead of visiting other nodes first**
  (there is never an advantage through reduced edge weight).

Make A* terminate earlier than for general graphs:

- no need to revisit nodes (path only gets longer)

- when the goal is directly reachable, there can be no other shorter path to it -> terminate

- not all neighbours of the current node have to be checked like in vanilla A* before continuing to the next node

3.3 Comparison to pyvisgraph

This package is similar to pyvisgraph which uses Lee's algorithm.

Pros:

- very reduced visibility graph (time and memory!)

- algorithms optimized for path finding

- possibility to convert and use grid worlds

Cons:

- parallel computing not supported so far

- no existing speed comparison

3.4 Contact

Tell me if and how your are using this package. This encourages me to develop and test it further.

Most certainly there is stuff I missed, things I could have optimized even further or explained more clearly, etc. I would be really glad to get some feedback.

If you encounter any bugs, have suggestions etc. do not hesitate to **open an Issue** or **add a Pull Requests** on Git. Please refer to the *contribution guidelines*

3.5 References

[1] Vinther, Anders Strand-Holm, Magnus Strand-Holm Vinther, and Peyman Afshani. "Pathfinding in Two-dimensional Worlds". no. June (2015).

3.6 Further Reading

Open source C++ library for 2D floating-point visibility algorithms, path planning: https://karlobermeyer.github.io/VisiLibity1/

Python binding of VisiLibity: https://github.com/tsaoyu/PyVisiLibity

Paper about Lee's algorithm: http://www.dav.ee/papers/Visibility_Graph_Algorithm.pdf

C implementation of Lee's algorithm: https://github.com/davetcoleman/visibility_graph

3.7 Acknowledgements

Thanks to:

- Georg Hess for improving the package in order to allow intersecting polygons.

- Ivan Doria for adding the command line interface.

API DOCUMENTATION

4.1 PolygonEnvironment

class extremitypathfinder.**PolygonEnvironment**

> Bases: object
>
> Class allowing to use polygons to represent "2D environments" and use them for path finding.
>
> Keeps a "loaded" and prepared environment for consecutive path queries. Internally uses a visibility graph optimised for shortest path finding. General approach and some optimisations theoretically described in: [1] Vinther, Anders Strand-Holm, Magnus Strand-Holm Vinther, and Peyman Afshani. "Pathfinding in Two-dimensional Worlds"
>
> TODO document parameters
>
> **prepared: bool = False**
>
> **holes: List[ndarray]**
>
> **extremity_indices: ndarray**
>
> **reprs_n_distances: Dict[int, ndarray]**
>
> **graph: Graph**
>
> **temp_graph: Graph | None = None**
>
> **boundary_polygon: ndarray**
>
> **coords: ndarray**
>
> **edge_vertex_idxs: ndarray**
>
> **extremity_mask: ndarray**
>
> **vertex_edge_idxs: ndarray**
>
> **property nr_edges: int**
>
> **property all_extremities: List[Tuple]**
>
> **property all_vertices: List[Tuple]**

store(*boundary_coordinates: ndarray | List[Tuple[float | int, float | int]]*, *list_of_hole_coordinates: ndarray | List[Tuple[float | int, float | int]]*, *validate: bool = False*)

saves the passed input polygons in the environment

Note: the passed polygons must meet these requirements:

- given as numpy or python array of coordinate tuples: `[(x1,y1), (x2,y2,)...]`
- no repetition of the first point at the end
- at least 3 vertices (no single points or lines allowed)
- no consequent vertices with identical coordinates in the polygons (same coordinates allowed)
- no self intersections
- edge numbering has to follow these conventions: boundary polygon counter clockwise, holes clockwise

---

Parameters

- **boundary_coordinates** – array of coordinates with counter clockwise edge numbering
- **list_of_hole_coordinates** – array of coordinates with clockwise edge numbering
- **validate** – whether the requirements of the data should be tested

Raises
    AssertionError – when validate=True and the input is invalid.

store_grid_world(*size_x: int*, *size_y: int*, *obstacle_iter: Iterable[Tuple[float | int, float | int]]*, *simplify: bool = True*, *validate: bool = False*)

Convert a grid-like into a polygon environment and save it

Prerequisites: grid world must not have single non-obstacle cells which are surrounded by obstacles ("white cells in black surrounding" = useless for path planning)

Parameters

- **size_x** – the horizontal grid world size
- **size_y** – the vertical grid world size
- **obstacle_iter** – an iterable of coordinate pairs (x,y) representing blocked grid cells (obstacles)
- **validate** – whether the input should be validated
- **simplify** – whether the polygons should be simplified or not. reduces edge amount, allow diagonal edges

export_pickle(*path: str = 'environment.pickle'*)

prepare()

Computes a visibility graph optimized (=reduced) for path planning and stores it

Computes all directly reachable extremities based on visibility and their distance to each other pre-procesing of the map. pre-computation for faster shortest path queries optimizes graph further at construction time

NOTE: initialise the graph with all extremities. even if a node has no edges (visibility to other extremities, dangling node), it must still be included!

Note: Multiple polygon vertices might have identical coords_rel. They must be treated as distinct vertices here, since their attached edges determine visibility. In the created graph however, these nodes must be merged at the end to avoid ambiguities!

Note: Pre computing the shortest paths between all directly reachable extremities and storing them in the graph would not be an advantage, because then the graph is fully connected. A* would visit every node in the graph at least once (-> disadvantage!).

---

within_map(*coords: ndarray*) → bool

> checks if the given coordinates lie within the boundary polygon and outside of all holes
>
> > **Parameters**
> > > **coords** – numerical tuple representing coordinates
> >
> > **Returns**
> > > whether the given coordinate is a valid query point

get_visible_idxs(*origin: int*, *candidates: Iterable[int]*, *coords: ndarray*, *vert_idx2repr: ndarray*, *vert_idx2dist: ndarray*) → Set[int]

find_shortest_path(*start_coordinates: Tuple[float | int, float | int]*, *goal_coordinates: Tuple[float | int, float | int]*, *free_space_after: bool = True*, *verify: bool = True*) → Tuple[List[Tuple[float, float]], float | None]

> computes the shortest path and its length between start and goal node
>
> > **Parameters**
> >
> > - **start_coordinates** – a (x,y) coordinate tuple representing the start node
> >
> > - **goal_coordinates** – a (x,y) coordinate tuple representing the goal node
> >
> > - **free_space_after** – whether the created temporary search graph graph should be deleted after the query
> >
> > - **verify** – whether it should be checked if start and goal points really lie inside the environment. if points close to or on polygon edges should be accepted as valid input, set this to `False`.
> >
> > **Returns**
> > > a tuple of shortest path and its length. ([], None) if there is no possible path.

4.2 PlottingEnvironment

PERFORMANCE

5.1 Speed Benchmark Results

obtained on MacBook Pro (15-inch, 2017), 2,8 GHz Intel Core i7 and *extremitypathfinder* version `2.4.1` using the
script `scripts/speed_benchmarks.py`

```
speed_benchmarks.py::test_env_preparation_speed PASSED                    [ 50%]
avg. environment preparation time 7.4e-03 s/run, 1.4e+02 runs/s
averaged over 1,000 runs


speed_benchmarks.py::test_query_speed PASSED                              [100%]
avg. query time 7.1e-04 s/run, 1.4e+03 runs/s
averaged over 1,000 runs
```

CONTRIBUTION GUIDELINES

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs via Github Issues.

If you are reporting a bug, please include:

- Your version of this package, python and Numba (if you use it)
- Any other details about your local setup that might be helpful in troubleshooting, e.g. operating system.
- Detailed steps to reproduce the bug.
- Detailed description of the bug (error log etc.).

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "help wanted" and not assigned to anyone is open to whoever wants to implement it - please leave a comment to say you have started working on it, and open a pull request as soon as you have something working, so that Travis starts building it.

Issues without "help wanted" generally already have some code ready in the background (maybe it's not yet open source), but you can still contribute to them by saying how you'd find the fix useful, linking to known prior art, or other such help.

6.1.4 Write Documentation

Probably for some features the documentation is missing or unclear. You can help with that!

6.1.5 Submit Feedback

The best way to send feedback is to file an issue via Github Issues.

If you are proposing a feature:

- Explain in detail how it would work.

- Keep the scope as narrow as possible, to make it easier to implement. Create multiple issues if necessary.

- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up this package for local development.

- Fork this repo on GitHub.

- Clone your fork locally

- To make changes, create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

- Check out the instructions and notes in `publish.py`

- Install `tox` and run the tests:

```
$ pip install tox
$ tox
```

The `tox.ini` file defines a large number of test environments, for different Python etc., plus for checking codestyle. During development of a feature/fix, you'll probably want to run just one plus the relevant codestyle:

```
$ tox -e codestyle
```

- Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

- Submit a pull request through the GitHub website. This will trigger the Travis CI build which runs the tests against all supported versions of Python.

CHANGELOG

7.1 2.7.2 (2024-04-10)

- added support for python 3.12

internal:

- added tests for python 3.12
- use ruff pre-commit hooks
- made dependency groups docs and plot optional
- added tox tes for documentation build

7.2 2.7.1 (2023-05-16)

internal:

- JIT compile more utility functions (including numpy.linalg.solve)
- add scipy dependency for JIT compiling numpy.linalg.solve
- updated supported python versions to ">=3.8,<3.12" (required by scipy)
- remove debug print statements and assertions

7.3 2.7.0 (2023-06-08)

- optional Numba JIT compilation offering significant speedup
- extra: *pip install extremitypathfinder[numba]*

7.4 2.6.0 (2023-06-04)

internal:

- implemented an optimised visibility graph algorithm: sort edges and candidates after their representation to always only check the relevant fraction of candidates for each edge. Runtime complexity $O(n^2 \log_2 n)$.
- added visibility computation tests
- automatically skip GitHub actions publishing when the version already exists. useful for minor improvements without publishing a version. build would always fail otherwise
- updated pinned dependencies to fix security alerts
- minor code refactoring

7.5 2.5.0 (2023-05-05)

- removed need for separate `.prepare()` call. Storing environment boundary data automatically triggers the preparation of the visibility graph. This is a non-breaking change. The `.prepare()` method is still available, but it is not needed anymore.

internal:

- updated dependency specification: networkx>=3, relaxed development dependency version requirements
- included tests for python 3.11
- minor code refactoring

7.6 2.4.1 (2022-08-22)

- bugfix: catch the case where no path is possible in the graph in the `networkx` A* implementation
- added speed benchmarks and performance section in the documentation with benchmark results

internal:

- optimisation: checking edges with the biggest angle range first
- optimisation: skipping visibility checks for the last extremity
- using optimised point in polygon check algorithm
- using undirected Graph: The precomputed graph usually makes up the majority of the visibility graph (in comparison to the temporarily added edges for query start and goal nodes) and this precomputed part has to be undirected. Use undirected graph everywhere.
- added test cases

7.7 2.4.0 (2022-08-18)

- A* and graph representation based on `networkx` library -> new dependency

7.8 2.3.0 (2022-08-18)

- major overhaul of all functionality from OOP to functional programming/numpy based

internal:

- added test cases

7.9 2.2.3 (2022-10-11)

- reverting changes of version `2.2.2`

7.10 2.2.2 (2022-07-10)

- [DEPRECATED]

7.11 2.2.1 (2022-07-10)

- packaging completely based on `pyproject.toml` (poetry)
- CI/CD: automatic publishing based on GitHub Actions

7.12 2.2.0 (2021-01-25)

- Included a command line interface
- Improved testing routines and codestyle

7.13 2.1.0 (2021-01-07)

IMPORTANT BUGFIX: in some cases the visibility computation was faulty (fix #23)

- added new test case

7.14 2.0.0 (2020-12-22)

- IMPROVEMENT: Different polygons may now intersect each other. Thanks to Georg Hess!
- BUGFIX: Fixed a bug that caused "dangling" extremities in the graph to be left out
- `TypeError` and `ValueError` are being raised instead of `AssertionError` in case of invalid input parameters with `validate=True`. Thanks to Andrew Costello!

7.15 1.5.0 (2020-06-18)

- BUGFIX: fix #16. introduce unique ordering of A* search heap queue with custom class `SearchState` (internal)

7.16 1.4.0 (2020-05-25)

- BUGFIX: fix clockwise polygon numbering test (for input data validation, mentioned in #12)

7.17 1.3.0 (2020-05-19)

- FIX #11: added option `verify` to `find_shortest_path()` for skipping the 'within map' test for goal and start points

7.18 1.2.0 (2020-05-18)

- supporting only python 3.7+
- fix #10: Memory leak in DirectedHeuristicGraph
- fix BUG where "dangling" extremities in the visibility graph would be deleted
- using generators to refer to the polygon properties (vertices,. . .) of an environment (save memory and remove redundancy)
- enabled plotting the test results, at the same time this is testing the plotting functionality
- added typing

internal:

- added sphinx documentation, included auto api documentation, improved docstrings
- added contribution guidelines
- add sponsor button
- updated publishing routine
- split up requirement files (basic, tests)
- specific tags for supported python versions in wheel
- testing all different python versions with tox
- added coverage tests

- added editorconfig

- specify version in VERSION file

- added new tests

7.19 1.1.0 (2018-10-17)

- optimised A*-algorithm to not visit all neighbours of the current node before continuing

7.20 1.0.0 (2018-10-07)

- first stable public version

7.21 0.0.1 (2018-09-27)

- birth of this package

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

V

W