Nonograms, Steps

Raphaël Parment

January 13, 2015

1 Polygonal lines approximation

1.1 Intro and Bookkeeping

The picture hidden in the Nonogram is that of a top view of an airplane (similar to a pictogram). I am only using polygonal lines, thus avoiding any non linear geometry (only straight lines). The input file is `point_files/points_plane.txt`. The input file is formated, such that each line represents four points, the first two being the x and y coordinates of the starting point of the line and the last two being the coordinates of the ending point of the line. Each line of text in the input file represents an actual geometric line between two points. These are the coordinates for the air plane model:

x_1	y_1	x_2	y_2
0	70	-10	50
-10	50	-10	20
-10	20	-50	0
-50	0	-50	-20
-50	-20	-10	-10
-10	-10	-10	-30
-10	-30	-20	-40
-20	-40	20	-40
20	-40	10	-30
10	-30	10	-10
10	-10	50	-20
50	-20	50	0
50	0	10	20
10	20	10	50
10	50	0	70

1.2 Code

1.2.1 DCEL business

The code is located at `C:/Users/Raphael/Documents/_Master/Thesis/notes/libraries`, which is thought of as the root directory for all code. I am using a free DCEL Python implementation

found at: `http://euler.slu.edu/~goldwasser/493/assignments/hw02/files`. It consists of 10 files:

- Point.py
  used to represent a 2D or 3D point
- Rational.py
  class used to create Points
- DCEL.py
  main module, with Vertex, Edge, Face, DCEL (those are the 4 main) classes
- geomutil.py
  module with geometric procedures implemented
- intersections.py
  module to check line intersections
- BinarySearchTree.py
- cs1graphics.py
- driver.py
- RedBlackTree.py
- triangulate.py
  this module is not used!

In order to create a DCEL object for a non simple polygon, I call the function *buildGeneral-Subdivision* located in the intersection module. If the polygon is simple, I call *BuildSimplePolygon* from the DCEL module. The returned object is a DCEL object taking care of all vertices, edges and faces. I added a couple of method to the DCEL class in the DCEL module: **plot_whole_dcel**, to the Face class: **walk_along** to walk along a face, **get_area** to compute the area of a face, **plot_face** to plot a face, **get_ordered_vertices** to return an ordered set of the vertices bounding a face.

1.2.2 Extension of lines

The drawing of the air plane is surrounded by a bounding box of length x pixels. In order to have somewhat of a complete Nonogram, I extended all the lines from the original drawing to the bounding box. Therefore we end up with a set of lines within which the original plane lies, however it is not easily recognisable. The extension of each line was done in the script `intersections_experiments.py`.

1.2.3 cPickle

Using the DCEL and related modules[1] we can create a doubly connected edge list for a set of edges, each described as a four coordinates in the input file. To speed up our computation, we do

[1] described in section 1.2.1

not want to re-build the DCEL everytime the script is run, therefore we need to save the created instance of the DCEL class. To do that we use the cPickle module. Due to high recursivity of the DCEL instance, we may not (did not work with Pickle, maybe try again with cPickle) *pickle* (save) it as a whole. However we truly only need the Vertices, Edges and Faces and the appropriate methods defined in those classes. We can pickle these parts of the instance, to be *unpickled* loaded in later scripts to heavily fasten running times. Here is the pickling and unpickling procedure:

pickling

```
import cPickle

# create a file
f_description = file(path_for_saving, 'wb')

# dumping the object into the file
cPickle.dump(object, f_description, protocol = cPickle.HIGHEST_PROTOCOL)

#close file
f_description.close()
```

unpickling

```
import cPickle

# fetch appropriate file
f_description = file(path_to_file, 'rb')

# unpickling/ loading
object_name = cPickle.load(f_description)

# close file
f_description.close()
```

In our main script we therefore want to firstly check if the three files (vertices.save, edges.save, faces.save) exists. If not, run the DCEL generating script and pickle the three objects. If they do exist, read the .save files and load the object into the current script.

Now we have two different script: `dcel_creation.py` used to create a dcel from the input file and then pickle the vertices, edges and faces into `structs/core_input_file_name_(vertices/edges/faces).save`. The second script is `nonograms_polygonal_lines.py`. It is the main program, in which we import `dcel_creation.py` and we unpickle (if needed) the pickle files: `structs/core_input_file_name_(vertices/edges/faces).save`.

1.3 GitHub

All the code is on github at: `https://github.com/Alledged/Nonograms-polygonal-lines`.