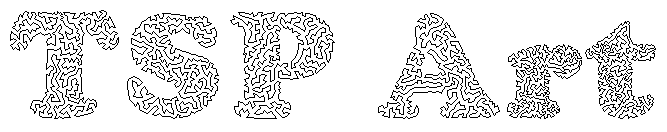
**Travelling Salesman Problem (TSP) Art in Python**



The intention of this repo is to provide a beginner-programmer-friendly way to enable people to make their own TSP Art using Python, where you can put in any image and generate a version made purely out of dots, and then another version that connects all these dots in a single continuous line where the end points meet each other.

**Outline of Algorithm**

There are two major steps to the algorithm:

1. Stippling (or 'pointillism') - the image is represented by small black dots of identical size in a way such that darker areas have more dots clustered closely together than lighter areas. The algorithm used here is called 'weighted Voronoi stippling'.
2. Determining and drawing the Travelling Salesman Problem Path - the [Travelling Salesman Problem](https://simple.wikipedia.org/wiki/Travelling_salesman_problem) is a classic mathematical optimization problem where given a list of locations, we are to find a single path that travels through all the locations only once and returns to the starting point. Here we use the dots drawn in the first step as our locations and use an algorithm to determine and then draw an appropriate path.

* [Python 3](https://www.python.org/downloads/) - you should also know how to use the console/command prompt, and run/execute a Python script. Note that command line options might be different for those using Anaconda.
* Optional: [Concorde TSP Solver](http://www.math.uwaterloo.ca/tsp/concorde/index.html)
* Optional: [Git](https://git-scm.com/)

Optional: Image editing program. Free/open-source ones: [Krita](https://krita.org/en/), [GIMP](https://www.gimp.org/), And lastly, the image(s) that you want to convert!

**Python Dependencies**

* Pillow (error )
* ortools (Note that this requires [Microsoft Visual C++ Redistributable for Visual Studio 2019, which can be found at the bottom of this link.](https://visualstudio.microsoft.com/downloads/?q=Visual+C%2B%2B+Redistributable+for+Visual+Studio))
* tqdm
* imageio
* scipy
* matplotlib

**What kind of images should I use for best results?**

**Format:** This will work for the common image formats (.jpg, .png). More obscure image formats might have some issues, so I'd recommend converting them to .jpg or .png first.

**Type**: Generally you'll want to use images that is a single object against a white background. Colour doesn't matter as much since the image is converted to grayscale as part of the stippling process.

Three images are provided for you in the images folder for you to practice on and to observe results. The scripts are initially configured to use smiley face-inverted.png in the images folder, and you can get an idea of what the output might look like if you check the example-output folder.

**1. Image Preprocessing (and potential problems!)**

Skip this step if you're a first timer. This step will only be relevant after you've run through a few images and want to tweak things a little, or have run into certain problems.

**Images with transparent backgrounds**: You'll want to colour these backgrounds white in your image-editing program, as some transparent backgrounds are set to black by default.

**Difficult to distinguish sections of similar colour/shade**: You may have seen this when trying out croissant-emoji.png. There's two ways to deal with this: one is to increase the number of dots available. The other is to increase the contrast or recolour sections appropriately using your image editing program.

**Compression noise/grain/artefacts**: Sometimes your initial image might not necessarily be smooth, or you'll see 'bits that aren't supposed to be there'. These vary a huge amount so there's no one surefire method for dealing with all of them. I know there are built-in methods in GIMP/Krita/Photoshop for dealing with them, but am no expert - usually the examples I work with are simple enough to manually clean using Brush and Fill tools.

**2. Stippling**

Open up stippling.py in the editor of your choice, and change the ORIGINAL\_IMAGE variable to the folder and image that you wish to stipple. So if our image is figure.png located in the images folder, you'd rename the variable to "images/figure.png"

What should happen on your first time:

* the console should show something similar to a progress bar, showing each iteration on a new line
* a window will pop up and show the dots arranging themselves
* closing aforementioned window will finish the script, and you should see two new files in the images folder:
  + figure-1024-stipple.png which is a stippled version of your original image, and
  + figure-1024-stipple.tsp which is a record of the coordinates of each of the points. This is the file we need for the next step.

Note: 1024 refers to the number of dots used, assuming you use the initial settings as given. If you change this number, the resulting filenames will also have their numbers changed. This is to make it easier to experiment with different numbers of dots without constantly having to delete the old files.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3. Acquiring & Drawing the TSP Solution**

**Using OR-Tools in Python**

Open draw-tsp-path.py in your editor, and change the variables as follows:

* ORIGINAL\_IMAGE should be the stippled image you obtained for Step 2: images/figure-1024-stipple.png
* IMAGE\_TSP should refer to the stipple .tsp file that is generated after Step 2: images/figure-1024-stipple.tsp

Run the file, wait for Python to do its job and when it's done, the final image will be generated as images/figure-1024-tsp.png.

**Using Concorde (Windows GUI)**

Open Concorde either by double clicking on figure-1024-stipple.tsp or opening the program separately and then loading figure-1024-stipple.tsp file into it. Concorde should then display a series of dots that should resemble what you see in figure-1024-stipple.png.

In the menu, click on 'Heuristics', select 'Lin Kernighan', then click OK. Concorde will then generate a tour that goes through all the points and returns to the starting point.

Save the tour as a file by selecting in the menu: File > Save Tour. In our example we'll save it as figure-tour.cyc.

Open draw-tsp-path-concorde.py in your editor and change the filenames at the top of the file

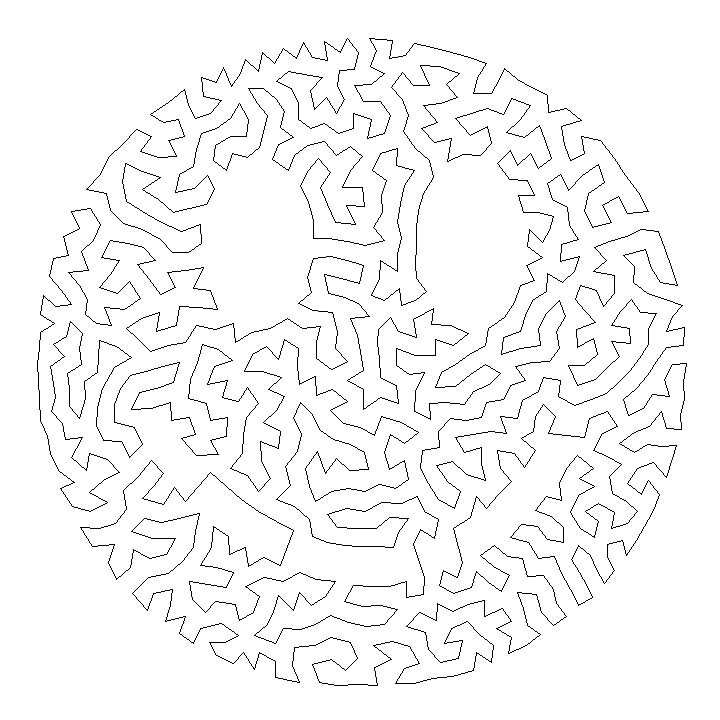
* ORIGINAL\_IMAGE should be the same initial image you used for Step 2: images/figure-1024-stipple.png
* IMAGE\_TSP should refer to the stipple .tsp file that is generated after Step 2: images/figure-1024-stipple.tsp
* IMAGE\_CYC should refer to the .cyc file that is generated from Concorde: images/figure-tour.cyc

Run the file and the script should generate the final image at images/figure-1024-tsp.png.

**When do I use Concorde over OR-Tools?**

Generally speaking you'll only want to use Concorde over OR-Tools if you have an image that has 'gaps' where you don't want a path to cross. Sometimes the OR-Tools algorithm may result in paths that 'cross-over' areas where you don't want them to, whereas the Concorde solver is less likely to achieve such a result.

To demonstrate as an example, in the example-output folder we have a smileyface-inverted.png:



Robert (Bob) Bosch ([Website](http://www.dominoartwork.com/), [Twitter](https://twitter.com/baabbaash/)) for the [original TSP art idea](http://www2.oberlin.edu/math/faculty/bosch/tspart-page.html). He's also written a book [Opt Art](https://www.amazon.com/Opt-Art-Mathematical-Optimization-Visual/dp/0691164061) which I highly recommend!

Adrian Secord ([Twitter](https://twitter.com/ajsecord)) for the weighted voronoi stippling algorithm. Nicholas Rougier ([Website](https://www.labri.fr/perso/nrougier/), [Twitter](https://twitter.com/NPRougier), [Github](https://github.com/rougier)) for the Python implementation.

See the Collection of Reference Links below for what I've found in my research while putting together this project.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Pillow (error )

**Python Imaging Library** is a [free and open-source](https://en.wikipedia.org/wiki/Free_and_open-source_software) additional [library](https://en.wikipedia.org/wiki/Library_(computing)) for the [Python programming language](https://en.wikipedia.org/wiki/Python_(programming_language)) that adds support for opening, [manipulating](https://en.wikipedia.org/wiki/Image_editing), and saving many different [image file formats](https://en.wikipedia.org/wiki/Image_file_formats). It is available for [Windows](https://en.wikipedia.org/wiki/Microsoft_Windows), Mac OS X and [Linux](https://en.wikipedia.org/wiki/Linux). The latest version of PIL is 1.1.7, was released in September 2009 and supports Python 1.5.2–2.7.

Development of the original project, known as **PIL**, was discontinued in 2011. Subsequently, a successor project named **Pillow** [forked](https://en.wikipedia.org/wiki/Fork_(software_development)) the PIL repository and added Python 3.x support. This fork has been adopted as a replacement for the original PIL in [Linux distributions](https://en.wikipedia.org/wiki/Linux_distribution) including [Debian](https://en.wikipedia.org/wiki/Debian_GNU/Linux) and [Ubuntu](https://en.wikipedia.org/wiki/Ubuntu)