Experimental Analyses of Heuristics for Horsefly-type Problems

Gaurish Telang

Contents

Part I

Overview

Chapter 1

Descriptions of Problems

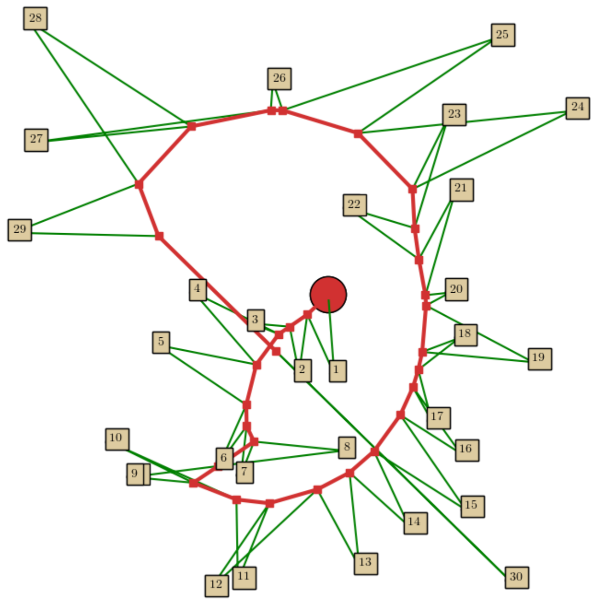


Figure 1.1: An Example of a classic Horsefly tour with $\varphi = 5$. The red dot indicates the initial position of the horse and fly, given as part of the input. The ordering of sites shown has been computed with a greedy algorithm which will be described later

The Horsefly problem is a generalization of the well-known Euclidean Traveling Salesman Problem. In the most basic version of the Horsefly problem (which we call **"Classic Horsefly"**), we are given a set of sites, the initial position of a truck(horse) with a drone(fly) mounted on top, and the speed of the drone-speed φ. [1] [2].

The goal is to compute a tour for both the truck and the drone to deliver package to sites as quickly as possible. For delivery, a drone must pick up a package from the truck, fly to the site and come back to the truck to pick up the next package for delivery to another site. [3] Both the truck and drone must coordinate their motions to minimize the time it takes for all the sites to get their packages. Figure 1.1 gives an example of such a tour computed using a greedy heuristic for $\varphi = 5$.

This suite of programs implement several experimental heuristics, to solve the above NP-hard problem and some of its variations approximately. In this short chapter, we give a description of the problem variations that we will be tackling. Each of the problems, has a corresponding chapter in Part 2, where these heuristics are described and implemented. We also give comparative analyses of their experimental performance on various problem instances.

Classic Horsefly This problem has already described in the introduction.

Segment Horsefly In this variation, the path of the truck is restricted to that of a segment, which we can consider without loss of generality to be $[0, 1]$. All sites, without loss of generality lie in the upper-half plane $\mathbb{R}^2_+$.

[1] The speed of the truck is always assumed to be 1 in any of the problem variations we will be considering in this report.
[2] φ is also called the "speed ratio".
[3] The drone is assumed to be able to carry at most one package at a time

Fixed Route Horsefly This is the obvious generalization of Segment Horsefly, where the path which the truck is restricted to travel is a piece-wise linear polygonal path. [4] Both the initial position of the truck and the drone are given. The sites to be serviced are allowed to lie anywhere in $\mathbb{R}^2$. Two further variations are possible in this setting, one in which the truck is allowed reversals and the other in which it is not.

One Horse, Two Flies The truck is now equipped with two drones. Otherwise the setting, is exactly the same as in classic horsefly. Each drone can carry only one package at a time. The drones must fly back and forth between the truck and the sites to deliver the packages. We allow the possibility that both the drones can land at the same time and place on the truck to pick up their next package. [5]

Reverse Horsefly In this model, each site (not the truck!) is equipped with a drone, which fly *towards* the truck to pick up their packages. We need to coordinate the motion of the truck and drone so that the time it takes for the last drone to pick up its package (the "makespan") is minimized.

Bounded Distance Horsefly In most real-world scenarios, the drone will not be able to (or allowed to) go more than a certain distance R from the truck. Thus with the same settings as the classic horsefly, but with the added constraint of the drone and the truck never being more than a distance R from the truck, how would one compute the truck and drone paths to minimize the makespan of the deliveries?

Watchman Horsefly In place of the TSP, we generalize the Watchman route problem here. [6] We are given as input a simple polygon and the initial position of a truck and a drone. The drone has a camera mounted on top which is assumed to have 360° vision. Both the truck and drone can move, but the drone can move at most euclidean distance [7] R from the truck.

We want every point in the polygon to be seen by the drone at least once. The goal is to minimize the time it takes for the drone to be able to see every point in the simple polygon. In other words, we want to minimize the time it takes for the drone (moving in coordinattion with the truck) to patrol the entire polygon.

[4] More generally, the truck will be restricted to travelling on a road network, which would typically be modelled as a graph embedded in the plane.

[5] In reality, one of the drones will have to wait for a small amount of time while the other is retrieving its package. In a more realisting model, we would need to take into account this "waiting time" too.

[6] although abstractly, the Watchman route problem can be viewed as a kind of TSP

[7] The version where instead geodesic distance is considered is also interesting

Chapter 2

Installation and Use

To run these programs you will need to install Docker, an open-source containerization program that is easily installable on Windows 10[1], MacOS, and almost any GNU/Linux distribution. For a quick introduction to containerization, watch the first two minutes of `https://youtu.be/_dfLOzuIg2o`

The nice thing about Docker is that it makes it easy to run softwares on different OS'es portably and neatly side-steps the dependency hell problem (`https://en.wikipedia.org/wiki/Dependency_hell`.) The headache of installing different library dependencies correctly on different machines running different OS'es, is replaced **only** by learning how to install Docker and to set up an X-windows connection between the host OS and an instantiated container running GNU/Linux.

A. [*Get Docker*] For installation instrutions watch

GNU/Linux `https://youtu.be/KCckWweNSrM`

Windows `https://youtu.be/ymlWt1MqURY`

MacOS `https://youtu.be/MU8HUVlJTEY`

B. [*Download customized Ubuntu image*] `docker pull gtelang/ubuntu_customized` [2]

C. [*Clone repository*] `git clone gtelang/horseflies_literate.git`

D. [*Mount and Launch*]

For GNU/Linux Open up your favorite terminal emulator, such xterm and then

- Copy to clipboard the output of `xauth list`
- `cd horseflies_literate`
- ``docker run -it --name horsefly_container --net=host -e DISPLAY -v /tmp/.X11-unix -v `pwd`:/horseflies_mnt``
- `cd horseflies_mnt`
- `xauth add` *<paste-from-clipboard>*

For Windows I had to follow the instructions in `https://dev.to/darksmile92/run-gui-app-in-linux-Docker-container-on-windows-host-4kde` to be able to run graphical user applications

E. [*Run experiments*] If you want to run all the experiments as described in the paper again to reproduce the reported results on your machine, then run [3],
`python main.py --run-all-experiments`.

If you want to run a specific experiment, then run
`python main.py --run-experiment <experiment-name>`.

See Index for a list of all the experiments.

F. [*Test algorithms interactively*] If you want to test the algorithms in interactive mode (where you get to select the problem-type, mouse-in the sites on a canvas, set the initial position of the truck and drone and set

[1] You might need to turn on virtualization explicitly in your BIOS, after installing Docker as I needed to while setting Docker up on Windows. Here is a snapshot of an image when turning on Intel's virtualization technology through the BIOS: `https://images.techhive.com/images/article/2015/09/virtualbox_vt-x_amd-v_error04_phoenix-100612961-large.idge.jpg`

[2] The customized Ubuntu image is approximately 7 GB which contains all the libraries (e.g. CGAL, VTK, numpy, and matplotlib) that I typically use to run my research codes portably.On my home internet connection downloading this Ubuntu-image typically takes about 5 minutes.

[3] Allowing, of course, for differences between your machine's CPU and mine when it comes to reporting absolute running time

φ), run `python main.py --<problem-name>`. The list of problems are the same as that given in the previous chapter. The problem name consists of all lower-case letters with spaces replaced by hyphens.

Thus for instance "Watchman Horsefly" becomes `watchman-horsefly` and "One Horse Two Flies" becomes `one-horse-two-flies`.

To interactively experiment with different algorithms for, say, the Watchman Horsefly problem , type at the terminal `python main.py --watchman-horsefly`

If you want to delete the Ubuntu image and any associated containers run the command [4]

```
docker rm -f horsefly_container; docker rmi -f ubuntu_customized
```

That's it! Happy horseflying!

[4] the ubuntu image is 7GB afterall!

Part II

Programs

Chapter 3

Overview of the Code Base

All of the code has been written in Python 2.7 and tested using the standard CPython implementation of the language. In some cases, calls will be made to external C++ libraries (mostly CGAL and VTK) using SWIG (http://www.swig.org/) for speeding up a slow routine or to use a function that is not available in any existing Python package.

Source Tree

```
..
|-- src
|   |-- expts
|   |-- lib
|   |   |-- problem_classic_horsefly.py
|   |   |-- utils_algo.py
|   |   `-- utils_graphics.py
|   |-- tests
|   `-- Makefile
|-- tex
|   |-- directory-tree.tex
|   |-- horseflies-1.pdf
|   |-- horseflies.pdf
|   |-- horseflies.tex
|   |-- problem_classic_horsefly.py
|   `-- standard_settings.tex
|-- webs
|   |-- problem-classic-horsefly
|   |   |-- algo-bottom-up-split.web
|   |   |-- algo-dumb.web
|   |   |-- algo-greedy-incremental-insertion.web
|   |   |-- algo-greedy-nn.web
|   |   |-- algo-k2-means.web
|   |   |-- algo-local-search-swap.web
|   |   |-- lower-bound-phi-mst.web
|   |   `-- problem-classic-horsefly.web
|   |-- problem-fixed-route-horsefly
|   |   `-- problem-fixed-route-horsefly.web
|   |-- problem-one-horse-two-flies
|   |   `-- problem-one-horse-two-flies.web
|   |-- problem-reverse-horsefly
|   |   `-- problem-reverse-horsefly.web
|   |-- problem-segment-horsefly
|   |   `-- problem-segment-horsefly.web
|   |-- problem-watchman-horsefly
|   |   `-- problem-watchman-horsefly.web
|   |-- descriptions-of-problems.web
|   |-- horseflies.web
|   |-- installation-and-use.web
```

```
|   |-- overview-of-code-base.web
|   `-- utility-functions.web
|-- horseflies.pdf
|-- main.py
|-- todolist.org
`-- weave-tangle.sh

12 directories, 32 files
```

There are three principal directories

`webs/` This contains the source code for the entire project written in the nuweb format along with documents (mostly images) needed during the compilation of the LaTeX files which will be extracted from the `.web` files.

`src/` This contains the source code for the entire project "tangled" (i.e. extracted) from the `.web` files.

`tex/` This contains the monolithic `horseflies.tex` extracted from the `.web` files and a bunch of other supporing LaTeX files. It also contains the final compiled `horseflies.pdf` (the current document) which contains the documentation of the project, interwoven with code-chunks and cross-references between them along with the experimental results.

The files in `src` and `tex` should not be touched. Any editing required should be done directly to the `.web` files which should then be weaved and tangled using `weave-tangle.sh`.

The Main Files

3.2.1

A. [*main.py*] The file `main.py` in the top-level folder is the *entry-point* for running code. Its only job is to parse the command-line arguments and pass relevant information to the handler functions for each problem and experiment.

B. [*Algorithmic Code*] All such files are in the directory `src/lib/`. Each of the files with prefix "`problem_*`" contain implmentations of algorithms for one specific problem. For instance `problem_watchman_horsefly.py` contains algorithms for approximately solving the Watchman Horsefly problem.

Since Horsefly-type problems are typically NP-hard, an important factor in the subsequent experimental analysis will require, comparing an algorithm's output against good lower bounds. Each such file, will also have routines for efficiently computing or approximating various lower-bounds for the corresponding problem's OPT.

C. [*Experiments*] All such files are in the directory `src/expt/` Each of the files with prefix "`expt_*`" contain code for testing hypotheses regarding a problem, generating counter-examples or comparing the experimental performance of the algorithm implementations for each of the problems. Thus `expt_watchman_horsefly.py` contains code for performing experiments related to the Watchman Horsefly problem.

If you need to edit the source-code for algorithms or experiment you should do so to the `.web` files in the web directory. Every problem has a dedicated *folder* containing source-code for algorithms and experiments pertaining to that problem. Every algorithm and experiment has a dedicated `.web` file in these problem directories. Such files are all "tied" together using the file with prefix `problem-<problem-name>` in that same directory (i.e. the file acts as a kind of handler for each problem, that includes the algorithms and experiment `web` files with the `@i` macro.)

3.2.2 Let's define the `main.py` file now.

Each problem or experiment has a handler routine that effectively acts as a kind of "main" function for that module that does house-keeping duties by parsing the command-line arguments passed by main, setting up the canvas by calling the appropriate graphics routines and calling the algorithms on the input specified through the canvas.

"../main.py" 12≡

```
import sys
sys.path.append('src/lib')

import problem_classic_horsefly as chf
#import problem_segment_horsefly as shf
#import problem_one_horse_two_flies as oh2f

if __name__=="__main__":
     # Select algorithm or experiment
     if (len(sys.argv)==1):
          print "Specify the problem or experiment you want to run"
          sys.exit()

     elif sys.argv[1] == "--problem-classic-horsefly":
          chf.run_handler()

     elif sys.argv[1] == "--problem-segment-horsefly":
          shf.run_handler()

     elif sys.argv[1] == "--problem-one-horse-two-flies":
          oh2f.run_handler()

     else:
          print "Option not recognized"
          sys.exit()
```
◇

Support Files

A. [*Utility Files*] All such utility files are in the directory `src/lib/`. These files contain common utility functions for manipulating data-structures, plotting and graphics routines common to all horsefly-type problems. All such files have the prefix `utils_*`. These Python files are generated from the single `.web` file `utils.web` in the `web` subdirectory.

B. [*Tests*] All such files are in the directory `src/test/` To automate testing of code during implementations, tests for various routines across the entire code-base have been written in files with prefix `test_*`.

Every problem, utility, and experimental files in `src/lib` and `src/expts` has a corresponding test-file in this folder.

Chapter 4

Some (Boring) Utility Functions

We will be needing some utility functions, for drawing and manipulating data-structures which will be implemented in files separate from `problem_classic_horsefly.py`. All such files will be prefixed with the work `utils_`. Many of the important common utility functions are defined here; others will be defined on the fly throughout the rest of the report. This chapter just collects the most important of the functions for the sake of clarity of exposition in the later chapters.

Graphical Utilities

Here We will develop routines to interactively insert points onto a Matplotlib canvas and clear the canvas. Almost all variants of the horsefly problem will involve mousing in sites and the initial position of the horse and fly. These points will typically be represented by small circular patches. The type of the point will be indicated by its color and size e.g. intial position of truck and drone will typically be represented by a large red dot while and the sites by smaller blue dots.

Matplotlib has extensive support for inserting such circular patches onto its canvas with mouse-clicks. Each such graphical canvas corresponds (roughly) to Matplotlib figure object instance. Each figure consists of several Axes objects which contains most of the figure elements i.e. the Axes objects correspond to the “drawing area” of the canvas.

4.1.1 First we set up the axes limits, dimensions and other configuration quantities which will correspond to the “without loss of generality” assumptions made in the statements of the horsefly problems. We also need to set up the axes limits, dimensions, and other fluff. The following fragment defines a function which “normalizes” a drawing area by setting up the x and y limits and making the aspect ratio of the axes object the same i.e. 1.0. Since Matplotlib is principally a plotting software, this is not the default behavior, since scales on the x and y axes are adjusted according to the data to be plotted.

"../src/lib/utils_graphics.py" 13≡

```
from matplotlib import rc
from colorama import Fore
from colorama import Style
from scipy.optimize import minimize
from sklearn.cluster import KMeans
import argparse
import itertools
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pprint as pp
import randomcolor
import sys
import time

xlim, ylim = [0,1], [0,1]
```

```
def applyAxCorrection(ax):
      ax.set_xlim([xlim[0], xlim[1]])
      ax.set_ylim([ylim[0], ylim[1]])
      ax.set_aspect(1.0)
◇
```

File defined by 13, 14ab, 15.

4.1.2

Next, given an axes object (i.e. a drawing area on a figure object) we need a function to delete and remove all the graphical objects drawn on it.

"../src/lib/utils_graphics.py" 14a≡

```
def clearPatches(ax):
    # Get indices cooresponding to the polygon patches
    for index , patch in zip(range(len(ax.patches)), ax.patches):
        if isinstance(patch, mpl.patches.Polygon) == True:
            patch.remove()

    # Remove line patches. These get inserted during the r=2 case,
    # For some strange reason matplotlib does not consider line objects
    # as patches.
    ax.lines[:]=[]

    #pp.pprint (ax.patches) # To verify that none of the patches are
    # polyon patches corresponding to clusters.
    applyAxCorrection(ax)
◇
```

File defined by 13, 14ab, 15.

4.1.3

Now remove the patches which were rendered for each cluster Unfortunately, this step has to be done manually, the canvas patch of a cluster and the corresponding object in memory are not reactively connected.I presume, this behavioue can be achieved by sub-classing.

"../src/lib/utils_graphics.py" 14b≡

```
def clearAxPolygonPatches(ax):

    # Get indices cooresponding to the polygon patches
    for index , patch in zip(range(len(ax.patches)), ax.patches):
        if isinstance(patch, mpl.patches.Polygon) == True:
            patch.remove()

    # Remove line patches. These get inserted during the r=2 case,
    # For some strange reason matplotlib does not consider line objects
    # as patches.
    ax.lines[:]=[]

    # To verify that none of the patches
    # are polyon patches corresponding
    # to clusters.
    #pp.pprint (ax.patches)
    applyAxCorrection(ax)
◇
```

File defined by 13, 14ab, 15.

4.1.4 Now for one of the most important routines for drawing on the canvas! To insert the sites, we double-click the left mouse button and to insert the initial position of the horse and fly we double-click the right mouse-button.

Note that the left mouse-button corresponds to button 1 and right mouse button to button 3 in the code-fragment below.

"../src/lib/utils_graphics.py" 15≡

```
## Also modify to enter initial position of horse and fly
def wrapperEnterRunPoints(fig, ax, run):
    """ Create a closure for the mouseClick event.
    """
    def _enterPoints(event):
        if event.name      == 'button_press_event'      and \
           (event.button   == 1 or event.button == 3)  and \
           event.dblclick == True                       and \
           event.xdata    != None                       and \
           event.ydata    != None:

             if event.button == 1:
               newPoint = (event.xdata, event.ydata)
               run.sites.append( newPoint  )
               patchSize  = (xlim[1]-xlim[0])/140.0

               ax.add_patch( mpl.patches.Circle( newPoint,
                                                 radius = patchSize,
                                                 facecolor='blue',
                                                 edgecolor='black'   )  )
               ax.set_title('Points Inserted: ' + str(len(run.sites)), \
                             fontdict={'fontsize':40})


             if event.button == 3:
                 inithorseposn = (event.xdata, event.ydata)
                 run.inithorseposn = inithorseposn
                 patchSize  = (xlim[1]-xlim[0])/70.0

                 # TODO: remove the previous red patches,
                 # which containg ht eold position
                 # of the horse and fly. Doing this is
                 # slightly painful, hence keeping it
                 # for later
                 ax.add_patch( mpl.patches.Circle( inithorseposn,
                                                   radius = patchSize,
                                                   facecolor= '#D13131', #'red',
                                                   edgecolor='black'   )  )

             # It is inefficient to clear the polygon patches inside the
             # enterpoints loop as done here.
             # I have just done this for simplicity: the intended behaviour
             # at any rate, is
             # to clear all the polygon patches from the axes object,
             # once the user starts entering in MORE POINTS TO THE CLOUD
             # for which the clustering was just computed and rendered.
             # The moment the user starts entering new points,
             # the previous polygon patches are garbage collected.
             clearAxPolygonPatches(ax)
             applyAxCorrection(ax)
             fig.canvas.draw()

    return _enterPoints
```
◇

File defined by 13, 14ab, 15.

Algorithmic Utilities

4.2.1 Given a list of points $[p_0, p_1, p_2, ...p_{n-1}]$. the following function returns, $[p_1 - p_0, p_2 - p_1, ..., p_{n-1} - p_{n-2}]$ i.e. it converts the list of points into a consecutive list of numpy vectors. Points should be lists or tuples of length 2

"../src/lib/utils_algo.py" 16a≡

```
import numpy as np
import random
from colorama import Fore
from colorama import Style


def vector_chain_from_point_list(pts):
    """ Given a list of points [p0,p1,p2,....p(n-1)]
    Make it into a list of numpy vectors
    [p1-p0, p2-p1,...,p(n-1)-p(n-2)]

    Points should be lists or tuples of length 2
    """
    vec_chain = []
    for pair in zip(pts, pts[1:]):
        tail= np.array (pair[0])
        head= np.array (pair[1])
        vec_chain.append(head-tail)

    return vec_chain
```
◇

File defined by 16ab, 17abc.

4.2.2 Given a polygonal chain, an important computation is to calculate its length. Typically used for computing the length of the horse's and fly's tours.

"../src/lib/utils_algo.py" 16b≡

```
def length_polygonal_chain(pts):
    """ Given a list of points [p0,p1,p2,....p(n-1)]
    calculate the length of its segments.

    Points should be lists or tuples of length 2

    If no points or just one point is given in the list of
    points, then 0 is returned.
    """
    vec_chain = vector_chain_from_point_list(pts)

    acc = 0
    for vec in vec_chain:
        acc = acc + np.linalg.norm(vec)
    return acc
```
◇

File defined by 16ab, 17abc.

4.2.3

The following routine is useful on long lists returned from external solvers. Often point-data is given to and returned from these external routines in flattened form. The following routines are needed to convert such a "flattened" list into a list of points and vice versa.

"../src/lib/utils_algo.py" 17a≡

```
def pointify_vector (x):
    """ Convert a vector of even length
    into a vector of points. i.e.
    [x0,x1,x2,...x2n] -> [[x0,x1],[x2,x3],,..[x2n-1,x2n]]
    """
    if len(x) % 2 == 0:
        pts = []
        for i in range(len(x))[::2]:
            pts.append( [x[i],x[i+1]] )
        return pts
    else :
        sys.exit('List of items does not have an even length to be able to be pointifyed')

def flatten_list_of_lists(l):
    """ Flatten vector
      e.g.  [[0,1],[2,3],[4,5]] -> [0,1,2,3,4,5]
    """
    return [item for sublist in l for item in sublist]

◇
```

File defined by 16ab, 17abc.

4.2.4

Python's default print function prints each list on a single line. For debugging purposes, it helps to print a list with one item per line.

"../src/lib/utils_algo.py" 17b≡

```
def print_list(xs):
    """ Print each item of a list on new line
    """
    for x in xs:
        print x
◇
```

File defined by 16ab, 17abc.

4.2.5

The following routines are self-explanatory and are hence gathered into one chunk.

"../src/lib/utils_algo.py" 17c≡

```
def partial_sums( xs ):
    """
    List of partial sums
    [4,2,3] -> [4,6,9]
    """
```

```
    psum = 0
    acc = []
    for x in xs:
        psum = psum+x
        acc.append( psum )

    return acc

def are_site_orderings_equal(sites1, sites2):
    """
    For two given lists of points test if they are
    equal or not. We do this by checking the Linfinity
    norm.
    """

    for (x1,y1), (x2,y2) in zip(sites1, sites2):
        if (x1-x2)**2 + (y1-y2)**2 > 1e-8:

            print Fore.BLUE+ "Site Orderings are not equal"
            print sites1
            print sites2
            print '-------------------------' + Style.RESET_ALL
            return False

    return True

    print "\n\n\n\n---------------------"


def bunch_of_random_points(numpts):
    cluster_size = int(np.sqrt(numpts))
    numcenters   = cluster_size

    import scipy
    import random
    centers = scipy.rand(numcenters,2).tolist()

    scale = 4.0
    points = []
    for c in centers:
        cx = c[0]
        cy = c[1]

        sq_size      = min(cx,1-cx,cy, 1-cy)
        cluster_size = int(np.sqrt(numpts))
        loc_pts_x    = np.random.uniform(low=cx-sq_size/scale,
                                         high=cx+sq_size/scale,
                                         size=(cluster_size,))
        loc_pts_y    = np.random.uniform(low=cy-sq_size/scale,
                                         high=cy+sq_size/scale,
                                         size=(cluster_size,))

        points.extend(zip(loc_pts_x, loc_pts_y))

    num_remaining_pts = numpts - cluster_size * numcenters

    remaining_pts = scipy.rand(num_remaining_pts, 2).tolist()
    points.extend(remaining_pts)

    return points
```

◇

File defined by 16ab, 17abc.

Chapter 5

Classic Horsefly

Module Overview

5.1.1 All algorithms to solve the classic horsefly problems have been implemented in `problem_classic_horsefly.py`. The `run_handler` function acts as a kind of main function for this module. It is called from `main.py` to process the command-line arguments and run the experimental or interactive sections of the code.

"../src/lib/problem_classic_horsefly.py" 20a≡

```
⟨ Relevant imports for classic horsefly 20b ⟩
def run_handler():
    ⟨ Define key-press handler 21a ⟩
    ⟨ Set up interactive canvas 23b ⟩

⟨ Local data-structures for classic horsefly 24 ⟩
⟨ Local utility functions for classic horsefly 37a, ... ⟩
⟨ Algorithms for classic horsefly 26, ... ⟩
⟨ Plotting routines for classic horsefly 38 ⟩
◇
```

Defines: `run_handler` Never used.

Module Details

5.2.1

⟨ *Relevant imports for classic horsefly* 20b ⟩ ≡

```
from matplotlib import rc
from colorama import Fore
from colorama import Style
from scipy.optimize import minimize
from sklearn.cluster import KMeans
import argparse
import itertools
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pprint as pp
import randomcolor
import sys
import time
```

```
import utils_algo
import utils_graphics
◇
```

Fragment referenced in 20a.

5.2.2

The key-press handler function detects the keys pressed by the user when the canvas is in active focus. This function allows you to set some of the input parameters like speed ratio φ, or selecting an algorithm interactively at the command-line, generating a bunch of uniform or non-uniformly distributed points on the canvas, or just plain clearing the canvas for inserting a fresh input set of points.

⟨ *Define key-press handler* 21a ⟩ ≡

```
# The key-stack argument is mutable! I am using this hack to my advantage.
def wrapperkeyPressHandler(fig,ax, run):
       def _keyPressHandler(event):
           if event.key in ['i', 'I']:
                ⟨ Start entering input from the command-line 21b ⟩
           elif event.key in ['n', 'N', 'u', 'U']:
                ⟨ Generate a bunch of uniform or non-uniform random points on the canvas 22 ⟩
           elif event.key in ['c', 'C']:
                ⟨ Clear canvas and states of all objects 23a ⟩
       return _keyPressHandler
◇
```

Fragment referenced in 20a.
Defines: `wrapperkeyPressHandler` 23b.

5.2.3

⟨ *Start entering input from the command-line* 21b ⟩ ≡

```
phi_str = raw_input(Fore.YELLOW + \
          "Enter speed of fly (should be >1): " +\
           Style.RESET_ALL)
phi = float(phi_str)

algo_str = raw_input(Fore.YELLOW + \
          "Enter algorithm to be used to compute the tour:\n Options are:\n" +\
        "  (e)   Exact \n"                                   +\
        "  (t)   TSP   \n"                                   +\
        "  (tl)  TSP   (using approximate L1 ordering)\n"    +\
        "  (k)   k2-center   \n"                             +\
        "  (kl)  k2-center (using approximate L1 ordering)\n"  +\
        "  (g)   Greedy\n"                                   +\
        "  (gl)  Greedy (using approximate L1 ordering])\n"  +\
        "  (ginc) Greedy Incremental  "  +\
        Style.RESET_ALL)

algo_str = algo_str.lstrip()

# Incase there are patches present from the previous clustering, just clear them
utils_graphics.clearAxPolygonPatches(ax)

if   algo_str == 'e':
      horseflytour = \
             run.getTour( algo_dumb,
                          phi )
elif algo_str == 'k':
```

```
        horseflytour = \
               run.getTour( algo_kmeans,
                            phi,
                            k=2,
                            post_optimizer=algo_exact_given_specific_ordering)
        print " "
        print Fore.GREEN, answer['tour_points'], Style.RESET_ALL
  elif algo_str == 'kl':
        horseflytour = \
               run.getTour( algo_kmeans,
                            phi,
                            k=2,
                            post_optimizer=algo_approximate_L1_given_specific_ordering)
  elif algo_str == 't':
        horseflytour = \
               run.getTour( algo_tsp_ordering,
                            phi,
                            post_optimizer=algo_exact_given_specific_ordering)
  elif algo_str == 'tl':
        horseflytour = \
               run.getTour( algo_tsp_ordering,
                            phi,
                            post_optimizer= algo_approximate_L1_given_specific_ordering)
  elif algo_str == 'g':
        horseflytour = \
               run.getTour( algo_greedy,
                            phi,
                            post_optimizer= algo_exact_given_specific_ordering)
  elif algo_str == 'gl':
        horseflytour = \
               run.getTour( algo_greedy,
                            phi,
                            post_optimizer= algo_approximate_L1_given_specific_ordering)

  elif algo_str == 'ginc':
        horseflytour = \
               run.getTour( algo_greedy_incremental_insertion,
                            phi )

  else:
        print "Unknown option. No horsefly for you! ;-D "
        sys.exit()

  #print horseflytour['tour_points']
  plotTour(ax,horseflytour, run.inithorseposn, phi, algo_str)
  utils_graphics.applyAxCorrection(ax)
  fig.canvas.draw()
  ◇
```

Fragment referenced in 21a.
Uses: `algo_exact_given_specific_ordering` 28, `algo_greedy_incremental_insertion` 31, `plotTour` 38.

5.2.4 This chunk generates points uniformly or non-uniformly distributed in the unit square $[0,1]^2$ in the Matplotlib canvas. I will document the schemes used for generating the non-uniformly distributed points later. These schemes are important to test the effectiveness of the horsefly algorithms. Uniform point clouds do no highlight the weaknesses of sequencing algorithms as David Johnson implies in his article on how to write experimental algorithm papers when he talks about algorithms for the TSP.

⟨ *Generate a bunch of uniform or non-uniform random points on the canvas* 22 ⟩ ≡

```
  numpts = int(sys.argv[1])
```

```
run.clearAllStates()
ax.cla()

utils_graphics.applyAxCorrection(ax)
ax.set_xticks([])
ax.set_yticks([])

fig.texts = []

import scipy
if event.key in ['n', 'N']: # Non-uniform random points
        run.sites = utils_algo.bunch_of_random_points(numpts)
else : # Uniform random points
        run.sites = scipy.rand(numpts,2).tolist()

patchSize  = (utils_graphics.xlim[1]-utils_graphics.xlim[0])/140.0

for site in run.sites:
    ax.add_patch(mpl.patches.Circle(site, radius = patchSize, \
                 facecolor='blue',edgecolor='black' ))

ax.set_title('Points : ' + str(len(run.sites)), fontdict={'fontsize':40})
fig.canvas.draw()
◇
```

Fragment referenced in 21a.

5.2.5

⟨ *Clear canvas and states of all objects* 23a ⟩ ≡

```
run.clearAllStates()
ax.cla()

utils_graphics.applyAxCorrection(ax)
ax.set_xticks([])
ax.set_yticks([])

fig.texts = []
fig.canvas.draw()
◇
```

Fragment referenced in 21a.

5.2.6

⟨ *Set up interactive canvas* 23b ⟩ ≡

```
fig, ax =  plt.subplots()
run = HorseFlyInput()
#print run

ax.set_xlim([utils_graphics.xlim[0], utils_graphics.xlim[1]])
ax.set_ylim([utils_graphics.ylim[0], utils_graphics.ylim[1]])
ax.set_aspect(1.0)
ax.set_xticks([])
ax.set_yticks([])

mouseClick   = utils_graphics.wrapperEnterRunPoints (fig,ax, run)
```

```
fig.canvas.mpl_connect('button_press_event' , mouseClick )

keyPress     = wrapperkeyPressHandler(fig,ax, run)
fig.canvas.mpl_connect('key_press_event', keyPress   )
plt.show()
◇
```

Fragment referenced in 20a.
Uses: `HorseFlyInput` 24, `wrapperkeyPressHandler` 21a.

Local Data Structures

5.3.1 This class manages the input and the output of the result of calling various horsefly algorithms.

⟨ *Local data-structures for classic horsefly* 24 ⟩ ≡

```
class HorseFlyInput:
      def __init__(self, sites=[], inithorseposn=()):
           self.sites         = sites
           self.inithorseposn = inithorseposn


      def clearAllStates (self):
          """ Set the sites to an empty list and initial horse position
          to the empty tuple.
          """
          self.sites = []
          self.inithorseposn = ()


      def getTour(self, algo, speedratio, k=None, post_optimizer=None):
          """ This method runs an appropriate algorithm for calculating
          a horsefly tour. The list of possible algorithms are
          inside this module prefixed with 'algo_'

          The output is a dictionary of size 2, containing two lists,
          - Contains the vertices of the polygonal
            path taken by the horse
          - The list of sites in the order
            in which they are serviced by the tour, i.e. the order
            in which the sites are serviced by the fly.
          """

          if k==None and post_optimizer==None:
                return algo(self.sites, self.inithorseposn, speedratio)
          elif k == None:
                return algo(self.sites, self.inithorseposn, speedratio, post_optimizer)
          else:
                #print Fore.RED, self.sites, Style.RESET_ALL
                return algo(self.sites, self.inithorseposn, speedratio, k, post_optimizer)

      def __repr__(self):
          """ Printed Representation of the Input for HorseFly
          """
          if self.sites != []:
              tmp = ''
              for site in self.sites:
                  tmp = tmp + '\n' + str(site)
```

```
            sites = "The list of sites to be serviced are " + tmp
        else:
            sites = "The list of sites is empty"

        if self.inithorseposn != ():
            inithorseposn = "\nThe initial position of the horse is " + \
                             str(self.inithorseposn)
        else:
            inithorseposn = "\nThe initial position of the horse has not been specified"

        return sites + inithorseposn
◇
```

Fragment referenced in 20a.
Defines: `HorseFlyInput` 23b.

Now that all the boring boiler-plate and handler codes have been written, its finally time for algorithmic ideas and implementations! Every algorithm is given an algorithmic overview followed by the detailed steps woven together with the source code.

Any local utility functions, needed for algorithmic or graphing purposes are collected at the end of this chapter.

Algorithm: Dumb Brute force

5.4.1 Algorithmic Overview

5.4.2 Algorithmic Details

⟨ *Algorithms for classic horsefly* 26 ⟩ ≡

```
def algo_dumb(sites, horseflyinit, phi):
    """ For each of the n factorial ordering of sites
    find the ordering which gives the smallest horsefly
    tour length
    """

    tour_length_fn = tour_length(horseflyinit)

    best_tour = algo_exact_given_specific_ordering(sites, horseflyinit, phi)

    i = 0
    for sites_perm in list(itertools.permutations(sites)):
        print "Testing a new permutation ", i, " of the sites"; i = i + 1

        #tour_for_current_perm = algo_exact_given_specific_ordering (sites_perm, \
        #                                                             horseflyinit, phi)
        tour_for_current_perm = algo_exact_given_specific_ordering (sites_perm, \
                                                                   horseflyinit, phi)
        if tour_length_fn(utils_algo.flatten_list_of_lists(tour_for_current_perm ['tour_points']) ) \
         < tour_length_fn(utils_algo.flatten_list_of_lists(            best_tour ['tour_points']) ):
                best_tour = tour_for_current_perm

                print Fore.RED + "Found better tour!" + Style.RESET_ALL

    #print Fore.RED + "\nHorse Waiting times are ",\
    #       best_tour['horse_waiting_times'] , \
    #       Style.RESET_ALL
    return best_tour
```

◇

Fragment defined by 26, 27, 28, 31.
Fragment referenced in 20a.
Uses: `algo_exact_given_specific_ordering` 28, `tour_length` 37a.

Algorithm: Greedy—Nearest Neighbor

5.5.1 Algorithmic Overview

5.5.2 Before proceeding we give a special case of the classical horseflies problem, which we term "collinear-horsefly". Here the objective function is again to minimize the tour-length of the drone with the additional restriction that the truck must always be moving in a straight line towards the site on the line-segment joining itself and the site, while the drone is also restricted to travelling along the same line segment.

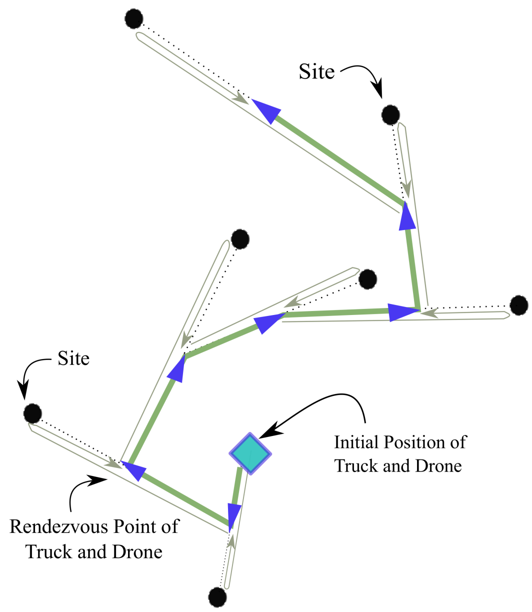


Figure 5.1: The Collinear Horsefly Problem

We can show that an optimal (unrestricted) horsfly solution can be converted to a collinear-horsefly solution at a constant factor increase in the makespan.

5.5.3 Algorithmic Details

⟨ *Algorithms for classic horsefly* 27 ⟩ ≡

```
def algo_greedy(sites, inithorseposn, phi, post_optimizer):
      """
      This implements the greedy algorithm for the canonical greedy
      algorithm for collinear horsefly, and then uses the ordering
      obtained to get the exact tour for that given ordering.

      Many variations on this are possible. However, this algorithm
      is simple and may be more amenable to theoretical analysis.

      We will need an inequality for collapsing chains however.
      """
      def next_rendezvous_point_for_horse_and_fly(horseposn, site):
            """
            Just use the exact solution when there is a single site.
            No need to use the collinear horse formula which you can
            explicitly derive. That formula is  an important super-special
            case however to benchmark quality of solution.
            """

            horseflytour = algo_exact_given_specific_ordering([site], horseposn, phi)
            return horseflytour['tour_points'][-1]

      # Begin the recursion process where for a given initial
      # position of horse and fly and a given collection of sites
      # you find the nearest neighbor proceed according to segment
```

```
        # horsefly formula for just and one site, and for the new
        # position repeat the process for the remaining list of sites.
        # The greedy approach can be extended to by finding the k
        # nearest neighbors, constructing the exact horsefly tour
        # there, at the exit point, you repeat by taking k nearest
        # neighbors and so on.
        def greedy(current_horse_posn, remaining_sites):
              if len(remaining_sites) == 1:
                    return remaining_sites
              else:
                    # For reference see this link on how nn queries are performed.
                    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.query.html
                    # Warning this is inefficient!!! I am rebuilding the
                    # kd-tree at each step. Right now, I am only doing
                    # this for convenience.
                    from scipy import spatial
                    tree = spatial.KDTree(remaining_sites)

                    # The next site to get serviced by the drone and horse
                    # is the one which is closest to the current position of the
                    # horse.
                    pts           = np.array([current_horse_posn])
                    query_result  = tree.query(pts)
                    next_site_idx = query_result[1][0]
                    next_site     = remaining_sites[next_site_idx]

                    next_horse_posn = \
                          next_rendezvous_point_for_horse_and_fly(current_horse_posn, next_site)
                    #print remaining_sites
                    remaining_sites.pop(next_site_idx) # the pop method modifies the list in place.

                    return [ next_site ] + greedy (current_horse_posn = next_horse_posn, \
                                                   remaining_sites    = remaining_sites)

        sites1 = sites[:]
        sites_ordered_by_greedy = greedy(inithorseposn, remaining_sites=sites1)

        # Use exact solver for the post optimizer step
        answer = post_optimizer(sites_ordered_by_greedy, inithorseposn, phi)
        return answer
    ◇
```

Fragment defined by 26, 27, 28, 31.
Fragment referenced in 20a.
Uses: `algo_exact_given_specific_ordering` 28.

5.5.4

⟨ *Algorithms for classic horsefly* 28 ⟩ ≡

```
    # ALGORITHMS FOR SINGLE HORSE SINGLE FLY SERVICING THE SITES IN THE GIVEN ORDER
    def algo_exact_given_specific_ordering (sites, horseflyinit, phi):
        """ Use the *given* ordering of sites to compute a good tour
        for the horse.
        """
        def ith_leg_constraint(i, horseflyinit, phi, sites):
            """ For the ith segment of the horsefly tour
            this function returns a constraint function which
            models the fact that the time taken by the fly
            is equal to the time taken by the horse along
```

```
    that particular segment.
    """
    if i == 0 :
        def _constraint_function(x):

            #print "Constraint  ", i
            start = np.array (horseflyinit)
            site  = np.array (sites[0])
            stop  = np.array ([x[0],x[1]])

            horsetime = np.linalg.norm( stop - start )

            flytime_to_site   = 1/phi * np.linalg.norm( site - start )
            flytime_from_site = 1/phi * np.linalg.norm( stop - site  )
            flytime           = flytime_to_site + flytime_from_site
            return horsetime-flytime

        return _constraint_function
    else :

        def _constraint_function(x):

           #print "Constraint  ", i
           start = np.array (  [x[2*i-2], x[2*i-1]]  )
           site  = np.array (  sites[i])
           stop  = np.array (  [x[2*i]  , x[2*i+1]]  )

           horsetime = np.linalg.norm( stop - start )

           flytime_to_site   = 1/phi * np.linalg.norm( site - start )
           flytime_from_site = 1/phi * np.linalg.norm( stop - site  )
           flytime           = flytime_to_site + flytime_from_site
           return horsetime-flytime

        return _constraint_function


def generate_constraints(horseflyinit, phi, sites):
    """ Given input data, of the problem generate the
    constraint list for each leg of the tour. The number
    of legs is equal to the number of sites for the case
    of single horse, single drone
    """
    cons = []
    for i in range(len(sites)):
        cons.append( { 'type':'eq',
                        'fun': ith_leg_constraint(i,horseflyinit,phi, sites) } )
    return cons



cons = generate_constraints(horseflyinit, phi, sites)
# Since the horsely tour lies inside the square,
# the bounds for each coordinate is 0 and 1
#x0 = np.empty(2*len(sites))
#x0.fill(0.5) # choice of filling vector with 0.5 is arbitrary

x0 = utils_algo.flatten_list_of_lists(sites) # the initial choice is just the sites
assert(len(x0) == 2*len(sites))
x0 = np.array(x0)

sol = minimize(tour_length(horseflyinit), x0, method= 'SLSQP', \
```

```
                                    constraints=cons, options={'maxiter':500})

    tour_points = utils_algo.pointify_vector(sol.x)

    # return the waiting times for the horse
    numsites            = len(sites)
    alpha               = horseflyinit[0]
    beta                = horseflyinit[1]
    s                   = utils_algo.flatten_list_of_lists(sites)
    horse_waiting_times = np.zeros(numsites)
    ps                  = sol.x
    for i in range(numsites):
        if i == 0 :
            horse_time         = np.sqrt((ps[0]-alpha)**2 + (ps[1]-beta)**2)
            fly_time_to_site   = 1.0/phi * np.sqrt((s[0]-alpha)**2 + (s[1]-beta)**2 )
            fly_time_from_site = 1.0/phi * np.sqrt((s[0]-ps[1])**2 + (s[1]-ps[1])**2)
        else:
            horse_time         = np.sqrt((ps[2*i]-ps[2*i-2])**2 + (ps[2*i+1]-ps[2*i-1])**2)
            fly_time_to_site   = 1.0/phi * np.sqrt(( (s[2*i]-ps[2*i-2])**2 + (s[2*i+1]-ps[2*i-1])**2 ))
            fly_time_from_site = 1.0/phi * np.sqrt(( (s[2*i]-ps[2*i])**2   + (s[2*i+1]-ps[2*i+1])**2 ))

        horse_waiting_times[i] = horse_time - (fly_time_to_site + fly_time_from_site)

    return {'tour_points'                : tour_points,
            'horse_waiting_times'        : horse_waiting_times,
            'site_ordering'              : sites,
            'tour_length_with_waiting_time_included': \
                                       tour_length_with_waiting_time_included(\
                                                    tour_points, \
                                                    horse_waiting_times,
                                                    horseflyinit)}
```

◇

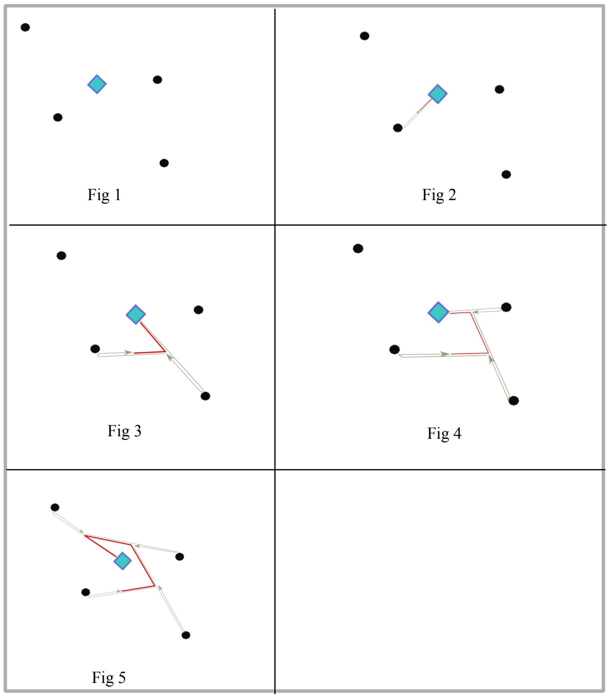
Fragment defined by 26, 27, 28, 31.
Fragment referenced in 20a.
Defines: `algo_exact_given_specific_ordering` 21b, 26, 27.
Uses: `tour_length` 37a, `tour_length_with_waiting_time_included` 37b.

Algorithm: Greedy—Incremental Insertion

Algorithmic Overview

5.6.1 The greedy nearest neighbor heuristic described in section 5.5 gives an $O(\log n)$ approximation for n sites in the plane. However, there exists an alternative greedy incremental insertion algorithm for the TSP that yields a 2-approximation. Similar to the greedy-nn algorithm we can generalize the greedy-incremental approach to the collinear-horseflies setting (cf: Figure 5.1).

5.6.2 In this approach, we maintain a list of visited sites V (along with the order of visitation $\mathcal{O}$) and the unvisited sites U. For the given collinear-horsefly tour serving V pick a site s from U along with a position in $\mathcal{O}$ (calling the resulting ordering $\mathcal{O}^{'}$) that minimizes the cost of the horsefly tour serving the sites $V \cup \{s\}$ in the order $\mathcal{O}^{'}$.



The figure above depicts the incremental insertion process for the case of 4 sites and $\varphi = 3$. The implementation of this algorithm for collinear-horseflies raises several interesting non-trivial data-structural questions in their own right: how to quickly find the site from U to insert into V, and keep track the changing length of the horsefly tour. Note that inserting a site causes the length of the tour of the truck to change, for all the sites after s.

Algorithmic Details

5.6.3

The implementation of the algorithm is "parametrized" over various strategies for insertion. i.e. we treat each insertion policy as a black-box argument to the function.

Efficient policies for detecting the exact or approximate point for cheapest insertion will be described in section 5.7. We also implement a "naive" policy as a way benchmark the quality and speed of implementation of future insertion policies.

⟨ *Algorithms for classic horsefly* 31 ⟩ ≡

```
    ⟨ Define auxiliary helper functions 34a, ... ⟩
    ⟨ Define various insertion policy classes 36a ⟩
    def algo_greedy_incremental_insertion(sites, inithorseposn, phi,
                                          insertion_policy_name = "naive",
                                          log_level             = None,
                                          write_io              = True,
                                          post_optimizer        = None):
          ⟨ Set log and input-output file config 32a ⟩
          ⟨ Set insertion policy class for current run 32b ⟩

          while insertion_policy.unvisited_sites:
             ⟨ Use insertion policy to find the cheapest site to insert into current tour 33a ⟩
             ⟨ Update list of visited and unvisited sites 33b ⟩
             ⟨ Write algorithm's current state to file 33c ⟩

          ⟨ Write input and output to file 33d ⟩
          ⟨ Return horsefly tour, along with additional information 33e ⟩
    ◇
```

Fragment defined by 26, 27, 28, 31.

Fragment referenced in 20a.
Defines: `algo_greedy_incremental_insertion` 21b.

5.6.4 Note that for each run of the algorithm, we create a dedicated directory and use a corresponding log file written as an AsciiDoc file written to that directory. It will typically containe detailed information on the progress of the algorithm and the steps executed. For the sake of neat formatting, the `.adoc` file will be converted to an HTML file via asciidoctor-latex.

For algorithm analysis, and verification of correctness,on the other hand, we will typically be interested in the states of the data-structures at the end of the while loop; each such state will be written out as a YAML file (that will also be accessible from the log file.) Such files can be useful for animating the algorithm.

Finally, just before returning the answer, we write the input and output to a separate YAML file. Thus all in all, there are three "types" of output files within each directory that corresponds to an algorithm's run: a log file, algorithm states files, and finally an input-output file.

⟨ *Set log and input-output file config* 32a ⟩ ≡

```
import sys, logging, datetime, os, errno

algo_name     = 'algo-greedy-incremental-insertion'
time_stamp    = datetime.datetime.now().strftime('Day-%Y-%m-%d_ClockTime-%H:%M:%S')
dir_name      = algo_name + '---' + time_stamp
log_file_name = dir_name + '/' + 'run.log'

# Create directory for writing data-files and logs to for
# current run of this algorithm
try:
    os.makedirs(dir_name)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

logging.basicConfig( filename = log_file_name,
                     level    = logging.DEBUG,
                     format   = '%(asctime)s: %(levelname)s: %(message)s',
                     filemode = 'w' )
logger = logging.getLogger()
logger.info("Started running greedy_incremental_insertion for classic horsefly")
```

◇

Fragment referenced in 31.

5.6.5 This fragment merely sets the variable `insertion_policy` to the appropriate function. This will later help us in studying the speed of the algorithm and quality of the solution for various insertion policies during the experimental analysis.

⟨ *Set insertion policy class for current run* 32b ⟩ ≡

```
if insertion_policy_name == "naive":
     insertion_policy = PolicyNaive(sites, inithorseposn, phi)
else:
     print insertion_policy_name
     sys.exit("Unknown insertion policy: " )

logger.debug("Finished setting insertion policy: " + insertion_policy_name)
```

```
sys.exit()
```
◇

Fragment referenced in 31.

5.6.6

Note that while defining the body of the algorithm, we treat the insertion policy (whose name has already been passed as an string argument) as a kind of black-box, since all policy classes have the same interface. The detailed implementation for the various insertion policies are given later.

⟨ *Use insertion policy to find the cheapest site to insert into current tour* 33a ⟩ ≡

```
pass
```
◇

Fragment referenced in 31.

5.6.7

⟨ *Update list of visited and unvisited sites* 33b ⟩ ≡

◇

Fragment referenced in 31.

5.6.8

⟨ *Write algorithm's current state to file* 33c ⟩ ≡

◇

Fragment referenced in 31.

5.6.9

⟨ *Write input and output to file* 33d ⟩ ≡

◇

Fragment referenced in 31.

5.6.10

⟨ *Return horsefly tour, along with additional information* 33e ⟩ ≡

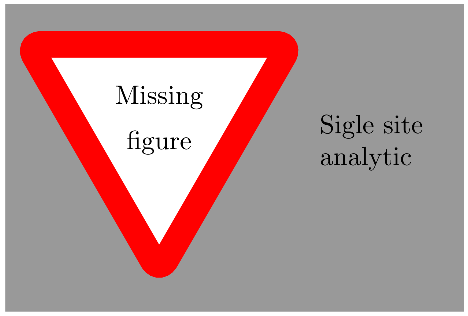
◇

Fragment referenced in 31.

5.6.11 We now define some of the functions that were referred to in the above chunks. Given the intial position of the truck and drone, and a list of sites, we need to compute the collinear horsefly tour length for the given ordering. This is the function that is used in every policy class while deciding which is the cheapest unvisited site to insert into the current ordering of visited sites.

Note that the order in which sites are passed to this function matters. It assumes that you want to compute the collinear horseflies tour length for the sites *in the given order*.

For this, we use two simple formulas for calculating the meeting point of the horse and fly and the distance the horse has to travel when there is only a single site.

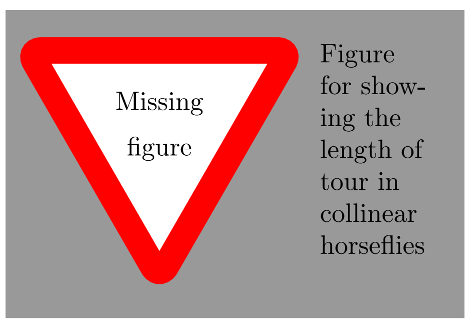


⟨ *Define auxiliary helper functions* 34a ⟩ ≡

```
def single_site_solution(site, horseposn, phi):
     pass # return rendezvous point and horse-travel length
◇
```

Fragment defined by 34ab.
Fragment referenced in 31.

With that the tour length functions for collinear horseflies can be implemented as an elementary instance of the fold pattern of functional programming. [1]



⟨ *Define auxiliary helper functions* 34b ⟩ ≡

```
def compute_collinear_horseflies_tour_length(sites, horseposn, phi):

     if not sites: # No more sites, left to visit!
          return 0

     else:         # Some sites are still left on the itinerary

          (rendezvous_pt, horse_travel_length) = single_site_solution(sites[0], horseposn, phi )
          return horse_travel_length  + \
                 compute_collinear_horseflies_tour_length( sites[1:], rendezvous_pt, phi )
◇
```

[1] Python has folds tucked away in some corner of its standard library. But I am not using it during the first hacky portion of this draft. Also Shane mentioned it has performance issues? Double-check this later!

Fragment defined by 34ab.
Fragment referenced in 31.
Defines: `compute_collinear_horseflies_tour_length` Never used.

Insertion Policies

We have finished implemented the entire algorithm, except for the implementation of the various insertion policy classes.

The main job of an insertion policy class is to keep track of the unvisited sites, the order of the visited sites and the horsefly tour itself. Every time, the method `.get_next_site(...)` is called, it chooses an appropriate (i.e. cheapest) unvisited site to insert into the current ordering, and update the set of visited and unvisited sites and details of the horsefly tour.

To do this quickly it will typically need auxiliary data-structures whose specifics will depend on the details of the policy chosen.

5.7.1 Naive Insertion

First, a naive implementation of the cheapest insertion heuristic, that will be useful in future benchmarking of running times and solution quality for implementations that are quicker but make more sophisticated uses of data-structures.

In this policy for each unvisited site we first find the position in the current tour, which after insertion into that position amongst the visited sites yields the smallest increase in the collinear-horseflies tour-length.

Then we pick the unvisited site which yields the overall smallest increase in tour-length and insert it into its computed position from its previous paragraph.

Clearly this implementation and has at least quadratic running time. Later on, we will be investigating algorithms and data-structures for speeding up this operation.

The hope is to be able to find a dynamic data-structure to perform this insertion in logarithmic time. Variations on tools such as the well-separated pair decomposition might help achieve this goal. Jon Bentley used kd-trees to perform the insertion in his experimental TSP paper, but he wasn't dealing with the shifting tour structure as we have in horseflies. Also he did not deal with the question of finding an approximate point for insertion. These

5.7.2

Since the interface for all policy classes will be the same, it is best, if have a base class for such classes. Since the details of the interface may change, I'll probably do this later. For now, I'll just keep all the policy classes completely separate while keeping the interface of the constructors and methods the same. I'll refactor things later.

The plan in that case should be to make an abstract class that has an abstract method called `insert\_unvisited\_site` and three data-fields made from the base-constructor named sites, inithorseposn and phi.

Classes which inherit this abstract base class, will add their own local data-members and methods for keeping track of data for insertion.

⟨ *Define various insertion policy classes* 36a ⟩ ≡

```
class PolicyNaive:

    def __init__(self, sites, inithorseposn, phi):

         # Remember input data for future processing

         self.sites          = sites
         self.inithorseposn  = inithorseposn
         self.phi            = phi

         # Initialize data-elements for whom I am
         # responsible for keeping track and manipulating.

         self.visited_sites   = []                # an index list that indexes into self.sites
         self.unvisited_sites = range(len(sites)) # Ditto
         self.horse_tour      = None              # A list of Steiner
                                                  # points where horse and fly meet

    ⟨ Methods for PolicyNaive 36b ⟩
```

◇

Fragment referenced in 31.

5.7.3

⟨ *Methods for* `PolicyNaive` 36b ⟩ ≡

```

def insert_unvisited_site(self):
      pass
```

◇

Fragment referenced in 36a.
Defines: `insert_unvisited_site` Never used.

Algorithm: Bottom-Up Split

5.8.1 Algorithmic Overview

5.8.2 Algorithmic Details

Algorithm: Local Search—Swap

5.9.1 Algorithmic Overview

5.9.2 Algorithmic Details

Algorithm: K2 Means

5.10.1 Algorithmic Overview

5.10.2 Algorithmic Details

Lower Bound: φ-MST

5.11.1 Overview

5.11.2 Computing the Lower-Bound

Local Utility Functions

5.12.1 For a given initial position of horse and fly return a function computing the tour length. The returned function computes the tour length in the order of the list of stops provided beginning with the initial position of horse and fly. Since the horse speed = 1, the tour length = time taken by horse to traverse the route.

This is in other words the objective function.

⟨ *Local utility functions for classic horsefly* 37a ⟩ ≡

```
def tour_length(horseflyinit):
   def _tourlength (x):

        # the first point on the tour is the
        # initial position of horse and fly
        # Append this to the solution x = [x0,x1,x2,....]
        # at the front
        htour = np.append(horseflyinit, x)
        length = 0

        for i in range(len(htour))[:-3:2]:
                length = length + \
                         np.linalg.norm([htour[i+2] - htour[i], \
                                         htour[i+3] - htour[i+1]])
        return length

   return _tourlength
```
◇

Fragment defined by 37ab.
Fragment referenced in 20a.
Defines: `tour_length` 26, 28, 38.

5.12.2 It is possible that some heuristics might return non-negligible waiting times. Hence I am writing a separate function which adds the waiting time (if it is positive) to the length of each link of the tour. Again note that because speed of horse = 1, we can add "time" to "distance".

⟨ *Local utility functions for classic horsefly* 37b ⟩ ≡

```
def tour_length_with_waiting_time_included(tour_points, horse_waiting_times, horseflyinit):
      tour_points   = np.asarray([horseflyinit] + tour_points)
      tour_links    = zip(tour_points, tour_points[1:])

      # the +1 because the inital position has been tacked on at the beginning
      # the solvers written the tour points except for the starting position
```

```
    # because that is known and part of the input. For this function
    # I need to tack it on for tour length
    assert(len(tour_points) == len(horse_waiting_times)+1)

    sum = 0
    for i in range(len(horse_waiting_times)):

        # Negative waiting times means drone/fly was waiting
        # at rendezvous point
        if horse_waiting_times[i] >= 0:
            wait = horse_waiting_times[i]
        else:
            wait = 0

        sum += wait + np.linalg.norm(tour_links[i][0] - tour_links[i][1], ord=2) #
    return sum
◇
```

Fragment defined by 37ab.
Fragment referenced in 20a.
Defines: `tour_length_with_waiting_time_included` 28, 38.

Plotting Routines

5.13.1

⟨ *Plotting routines for classic horsefly* 38 ⟩ ≡

```
def plotTour(ax,horseflytour, horseflyinit, phi, algo_str, tour_color='#d13131'):
    """ Plot the tour on the given canvas area
    """

    # Route for the horse
    xhs, yhs = [horseflyinit[0]], [horseflyinit[1]]
    for pt in horseflytour['tour_points']:
        xhs.append(pt[0])
        yhs.append(pt[1])

    # List of sites
    xsites, ysites = [], []
    for pt in horseflytour['site_ordering']:
        xsites.append(pt[0])
        ysites.append(pt[1])

    # Route for the fly. The fly keeps alternating
    # between the site and the horse
    xfs , yfs = [xhs[0]], [yhs[0]]
    for site, pt in zip (horseflytour['site_ordering'],
                         horseflytour['tour_points']):
        xfs.extend([site[0], pt[0]])
        yfs.extend([site[1], pt[1]])

    print "\n----------"
    print "Horse Tour"
    print "-----------"
    waiting_times = [0.0] + horseflytour['horse_waiting_times'].tolist() # the waiting time at the starting point
    #print waiting_times
```

```
    for pt, time in zip(zip(xhs,yhs), waiting_times) :
        print pt, Fore.GREEN, " ---> Horse Waited ", time, Style.RESET_ALL

    print "\n----------"
    print "Fly Tour"
    print "----------"
    for item, i in zip(zip(xfs,yfs), range(len(xfs))):
        if i%2 == 0:
           print item
        else :
           print Fore.RED + str(item) + "----> Site" +  Style.RESET_ALL

    print "----------------------------------"
    print Fore.GREEN, "\nSpeed of the drone was set to be", phi
    #tour_length = utils_algo.length_polygonal_chain( zip(xhs, yhs))
    tour_length = horseflytour['tour_length_with_waiting_time_included']
    print "Tour length of the horse is ",  tour_length
    print "Algorithm code-Key used "    , algo_str, Style.RESET_ALL
    print "----------------------------------\n"

    #kwargs = {'size':'large'}
    for x,y,i in zip(xsites, ysites, range(len(xsites))):
          ax.text(x, y, str(i+1), bbox=dict(facecolor='#ddcba0', alpha=1.0))
    ax.plot(xfs,yfs,'g-') # fly tour is green
    ax.plot(xhs, yhs, color=tour_color, marker='s', linewidth=3.0) # horse is red


    # Initial position of horse and fly
    ax.add_patch( mpl.patches.Circle( horseflyinit,
                                      radius = 1/34.0,
                                      facecolor= '#D13131', #'red',
                                      edgecolor='black'   )  )


    fontsize = 10
    tnrfont = {'fontname':'Times New Roman'}
    ax.set_title(  'Algorithm Used: ' + algo_str +  '\nTour Length: ' \
                    + str(tour_length)[:7], fontdict={'fontsize':fontsize}, **tnrfont)
    ax.set_xlabel('Number of sites: ' + str(len(xsites)) + '\nDrone Speed: ' + str(phi) ,
                  fontdict={'fontsize':fontsize}, **tnrfont)
```
◇

Fragment referenced in 20a.
Defines: `plotTour` 21b.
Uses: `tour_length` 37a, `tour_length_with_waiting_time_included` 37b.

Chapter Index of Fragments

Chapter Index of Identifiers

Chapter 6

Fixed Route Horsefly

Chapter 7

One Horse, Two Flies

Chapter 8

Reverse Horsefly

Chapter 9

Watchman Horsefly

Appendices

Appendix A

Index of Files

Appendix B

Man-page for `main.py`