Analyses of Experimental Heuristics for Package-Handoff Type Problems

Gaurish Telang

Contents

Page

1 Overview **3**

2 Single Package Handoff **5**

2.1 Wavefront Algorithms 5

2.2 Run Handler associated with this Chapter 10

2.3 Chapter Index of Fragments 12

2.4 Chapter Index of Identifiers 12

Appendices **14**

A Supporting Code **15**

A.1 Main File 15

A.2 Algorithmic Utilities 20

A.3 Graphical Utilities 21

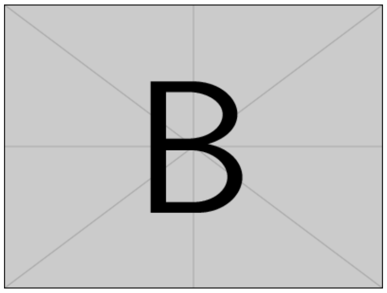
Chapter 1

Overview

How do you get a package from point A to point B with a fleet of carrier drones each capable of various maximum speeds? This is the question we try to answer in some of its various avatars by developing algorithms, heuristics, local optimality heuristics and lower-bounds.

Specifically, we are given as input the positions P_i of n drones (labelled 1 through n) in the plane each capable of a maximum speed of u_i. Also given is a package present at S that needs to get to T. Each drone is capable of picking up the package and flying with speed u_i to another point to hand the package off to another drone.

(a) An example of a carrier drone. Image taken from [1]



(b) A fleet of drones such as on the left, coordinating to move a package from S to T in the least time possible.

Figure 1.1: An instance of the Package Handoff problem for a single package

The challenge is to figure how to get the drones to cooperate to send the package from S to T in the least possible time i.e. minimize the makespan of the delivery process.

To solve the problem we need to be able to do several things

- Figure out which subset $S = \{i_1, i_2, \ldots i_k\}$ of the drones are used in the optimal schedule.
- Find the order in which the handoffs happend between the drones used in a schedule.
- Find the "handoff" points when drone i_m hands over the package to drone i_{m+1} for all $m \leq k-1$ [1]

This category of problems is a generalization of computing shortest paths in $\mathbb{R}^2$ between two points. As far as we know such problems have not been considered before in the operations research or computational geometry literature; it is, however, reminescent of the Weighted Region Problem [2] (henceforth abbreviated as WRP) where one needs to figure out how to compute a shortest *weighted* path between two points in the plane that has been partitioned into convex polygonal regions, each associated with a constant multiplicative weight for scaling the euclidean distance between two points *within* that region.

The distinctive feature of this problem and its generalizations is figuring out how to make multiple agents of *varying* capabilities located at different points in $\mathbb{R}^2$ (such as maximum capable speed, battery capacity, tethering constraints etc.) *cooperate* in transporting one or more packages most efficiently from their given sources to their target destinations.

While we are framing these problems in terms of drones, one can also apply this problem in routing a fleet of taxis to get passengers from their pickup to their dropoff locations. Interesting problems might arise in this scenario itself (e.g. what if the sequence of pickup and dropoff locations for passengers happen in an online manner, say when passengers request or cancel rides with their smartphones?) We leave the investigation of these latter fascinating problems for future work. All problems considered in this article are in the offline setting.

[1] The final drone i_k in the schedule flies with the package to the target site T

Each chapter in this document is devoted to developing algorithms for a specific variant of the package handoff problem (henceforth abbreviated as `PHO`), beginning with the plain-vanilla single package handoff problem described above. For most algorithms we will also be giving implementations in Python described in a literate-programming style [2] [3] using the NuWeb literate programming tool [4] for weaving and tangling the code-snippets.

You can check out the Package Handoff code from the following GitHub repository:

https://github.com/gtelang/PackageHandoff_Python

The `README` file in the repository gives instructions on how to run the code and any of the associated experiments.

[2] Which essentially means you will see code-snippets interleaved with the actual explanation of the algorithms. The code snippets are then extracted using a literate programming tool (using a so-called a "weaver" and "tangler") into an executable Python program

Chapter 2

Single Package Handoff

In this chapter, we consider the problem posed at the beginning of the Overview chapter. For convenience we state the problem again below

> *Given the positions P_i of n drones (labelled 1 through n) in $\mathbb{R}^2$ each capable of a maximum speed of $u_i \geq 0$. Also given is a package present at S that needs to get to T. Each drone is capable of picking up the package and flying with speed u_i to another point to hand the package off to another drone (which in turn hands the package off to another drone and so on).*
>
> *Find the best way to coordinate the movement of the drones to get the package from S to T in the least possible time i.e. minimize the makespan of the delivery process.*

Note that in the optimal schedule, it is easy to construct an example such that not all drones will necessarily participate in getting the package from S to T. The challenge is to figure out which subset of drones to use, along with the handoff points.

However, the following observations are crucial for the development of algorithms in this chapter.

Lemma 1.

1. *For the single delivery package handoff problem, a slower drone, always hands off the package to a faster drone, in any optimal schedule. Thus, once we know which drones participate in the schedule, the order in which they participate in the handoff from start to finish is determined according to their speeds, sorted from lowest to highest.* [a]
2. *All drones involved in the optimal schedule start moving at time $t = 0$ at full speed along a straight line towards a handoff point . The drones not involved can remain stationary since they don't participate in the package transport.*
3. *No drone waits on any other drone at the rendezvous points in any optimal schedule. i.e. if two drones rendezvous at some point H, they arrive at H are precisely the same time on the clock.*
4. *The path of the package is a monotonic piecewise straight polygonal curve with respect to the direction $\overrightarrow{ST}$ no matter what the intial positions P_i or speeds u_i of the drones.* [b]

[a] This property is unfortunately not true when there are multiple packages to be delivered to their respective desitinations, even for the case where the sources and targets for all the packages are the same. Examples where this happens are given in the next chapter.

[b] We conjecture this property to be true even for the case of multiple packages i.e. the path of travel of each package is monotonic with respect to the vectors S_iT_i's

Before proceeding, we first fix some notation:

- (P_i, u_i) for $1 \leq i \leq n$ where $P_i \in \mathbb{R}^2$ and $u_i \geq 0$, $S, T \in \mathbb{R}^2$ respectively stand for the initial positions, speed, and source and target points for a package.
- $(S = H_{i_0}), H_{i_1} \dots H_{i_k}$ for $0 \leq i_0, \dots i_k \leq n$ stand for points where the drones with labels $i_0, \dots i_k$ handle the package in that order. More precisely H_{i_j} is the point where drone i_{j-1} hands off the package to drone i_j for $1 \leq j \leq k$.

Wavefront Algorithms

The algorithms in this section are inspired by the Continuous Dijkstra paradigm used in computing shortest paths for the Weighted Region Problem and for computing euclidean shortest paths in the presence of polygonal obstacles [2, 5]. The approximation and locality properties of these heuristics are considered later in the chapter.

The general idea is simple: consider expanding circular wavelets centered at the positions P_i, each expanding with speed u_i. The drones invovled in the schedule are then calculated by observing how the wavelets interact in time. The various heuristics differ according to how the subset of drones involved in the delivery process is figured out based on nature of the "wavefront" used to keep track of the current tentative location of the package.

Once this subset of drones is calculated, we use convex optimization (via the convex optimization modelling language CVXPY [6]) to figure out *exactly* the handoff points for the drones involved in transporting the package from the source to the destination.

Precise details follow in the subsections below.

2.1.1 Preliminary Data Structures

Before proceeding, lets design some housekeeping data-structures to represent the problem. The following data-structure simply maintains the information about the drones, the source and target used as input to the problem. To get a PHO tour for the package, algorithms are passed as first class values to the method `get_tour` of this class.

Note that each algorithm does its own plotting and animation in a separate matplotlib window if so requested via the boolean flags `plot_tour_p` , and `animate_tour_p`. If both animation and plotting are requested they are done in separate windows each.

⟨ *PHO Data Structures* 6 ⟩ ≡

```
class Single_PHO_Input:
    def __init__(self, drone_info = [] , source = None, target=None):
           self.drone_info = drone_info
           self.source     = source
           self.target     = target

    def get_drone_pis (self):
           return [self.drone_info[idx][0] for idx in range(len(self.drone_info)) ]

    def get_drone_uis (self):
           return [self.drone_info[idx][1] for idx in range(len(self.drone_info)) ]

    def get_tour(self, algo):
           return algo( self.drone_info, self.source, self.target,
                        plot_tour_p = True)

    # Methods for \verb|ReverseHorseflyInput|
    def clearAllStates (self):
          self.drone_info = []
          self.source = None
          self.target = None
```

◇

Fragment defined by 6, 7a.
Fragment referenced in 15.
Defines: `Single_PHO_Input` 10.

2.1.2 One-Dimensional Greedy Wavefront

The followinf data-structure keeps track of a single wavelet expanding at uniform speed around a given center starting at a particular clock time.

⟨ *PHO Data Structures* 7a ⟩ ≡

```
class DroneWavelet:
    def __init__(self, startposn, speed, drone_idx,  clock_time):
          self.center          = startposn
          self.speed           = speed
          self.drone_label     = drone_idx
          self.last_reset_time = clock_time

    def get_center(self):
          return self.center

    def get_speed(self):
          return self.speed

    def get_associated_drone(self):
          return self.drone_label

    def wavelet_size(self, clock_time): # radius of the disk
           return (clock_time - last_reset_time) * self.speed

    def reset_wavelet(self, center, clock_time):
         self.center = center
         self.last_reset_time = clock_time

◇
```

Fragment defined by 6, 7a.
Fragment referenced in 15.

The following function simply calculates the time taken for a drone to move between two points at a given uniform speed.

⟨ *PHO Algorithms* 7b ⟩ ≡

```
def time_of_travel(start, stop, speed):
     start = np.asarray(start)
     stop  = np.asarray(stop)

     return np.linalg.norm(stop-start)/speed
◇
```

Fragment defined by 7bc, 9.
Fragment referenced in 15.

⟨ *PHO Algorithms* 7c ⟩ ≡

```
def algo_odw(drone_info, source, target,
             animate_tour_p = False,
             plot_tour_p    = False):


    from scipy.optimize import minimize

    print Fore.CYAN, "\n============================================================"
    print            "Running the one dimensional wavefront algorithm (algo_odw) "
    print            "============================================================", Style.RESET_ALL

    source = np.asarray(source)
    target = np.asarray(target)
    sthat  = (target-source)/np.linalg.norm(target-source) # unit vector pointing from source to target

    # Intialize wavelets for all drones
```

```
numdrones  = len(drone_info)
clock_time = 0.0

drone_wavelets = []
for (initposn, speed), idx in zip(drone_info, range(numdrones)):
     drone_wavelets.append(  DroneWavelet(initposn, speed, idx, clock_time)  )

# Find the drone which can get to the source the quickest
tmin = np.inf
imin = None
for idx in range(numdrones):
     initdroneposn = drone_info[idx][0]
     dronespeed    = drone_info[idx][1]
     tmin_idx = time_of_travel(initdroneposn, source, dronespeed)

     if tmin_idx < tmin:
         tmin = tmin_idx
         imin = idx

# Reset wavelet for the drone which reached the source the fastest
clock_time = tmin
drone_wavelets[imin].reset_wavelet(source, clock_time)

current_package_handler_idx = imin
current_package_position = source

drone_pool = range(numdrones)
drone_pool.remove(imin)
used_drones = []

package_reached_p   = False
while not(package_reached_p):
      time_to_target_without_help =\
          np.linalg.norm((target-current_package_position))/drone_info[current_package_handler_idx][1]
      tintercept_min     = np.inf
      idx_intercept_min  = None

      for idx in drone_pool:
          # Calculate tintercept here via scipy
          u1 = drone_info[current_package_handler_idx][1]
          u2 = drone_info[idx][1]

          if u2 < u1:
            continue # there won't be any use of the slower drone u2 in speeding up the package delivery process.
          else:
            p1 = np.asarray(drone_info[idx][0])
            p2 = np.asarray(drone_info[current_package_handler_idx][0])

            d  = np.linalg.norm( p1 - p2 )

            veca  = target-p1
            vecb  = p2 - p1
            costh = np.dot( veca, vecb ) / (np.linalg.norm(veca) * np.linalg.norm(vecb))

            fun    = lambda t: t[0]
            cons   = ({'type': 'ineq', 'fun': lambda t:  u2 + d - u1 * costh * (t[0]-clock_time)/t[0] },
                      {'type': 'ineq', 'fun': lambda t:  u2 - d + u1 * costh * (t[0]-clock_time)/t[0] })

            res = minimize(fun, [clock_time] , method='SLSQP', bounds=[(clock_time, None)], constraints=cons )

            if res.success:
                  print Fore.CYAN, "Yay! Solver converged in "      , res.nit
```

```
              else :
                    print Fore.RED, "Boo! Solver did not converge!", res.nit, " iterations."
                    print res.status
                    print res.message

              tintercept = res.x[0]

              print res.x, tintercept, tintercept_min
              print tintercept < tintercept_min
              if tintercept < tintercept_min:
                 tintercept_min    = tintercept
                 idx_intercept_min = idx


        if time_to_target_without_help < tintercept_min :
            package_reached_p = True
        else:

            package_handler_speed     = drone_info[current_package_handler_idx][1]
            current_package_position  = current_package_position + \
                 package_handler_speed * (tintercept_min - clock_time) *  sthat

            clock_time = tintercept_min

            current_package_handler_idx = idx_intercept_min
            drone_pool.remove(idx_intercept_min)
            used_drones.append(idx_intercept_min)

            drone_wavelets[current_package_handler_idx].reset_wavelet(current_package_position, clock_time)

  tour = algo_pho_exact_given_order_of_drones ( [drone_info[idx] for idx in used_drones],source,target )

  if plot_tour_p:
       print Fore.GREEN, "Plotting the computed tour", Style.RESET_ALL

  if animate_tour_p:
       print Fore.CYAN, "Animating the computed tour", Style.RESET_ALL

  return tour
◇
```

Fragment defined by 7bc, 9.
Fragment referenced in 15.
Defines: `algo_odw` 10.
Uses: `algo_pho_exact_given_order_of_drones` 9.

⟨ *PHO Algorithms* 9 ⟩ ≡

```
def  algo_pho_exact_given_order_of_drones ( drone_info ,source, target ):
    pass

◇
```

Fragment defined by 7bc, 9.
Fragment referenced in 15.
Defines: `algo_pho_exact_given_order_of_drones` 7c.

Run Handler associated with this Chapter

⟨ *PHO Run Handlers* 10 ⟩ ≡

```
def single_pho_run_handler():

    def wrapperEnterRunPoints(fig, ax, run):
      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:
                 # Insert blue circle representing the initial position of a drone
                 print Fore.GREEN
                 newPoint = (event.xdata, event.ydata)
                speed    = float(raw_input('What speed do you want to set for the drone @ ' + str(newPoint)) + '\n')
                 run.drone_info.append( (newPoint, speed) )
                 patchSize  = (xlim[1]-xlim[0])/40.0
                 print Style.RESET_ALL

                 ax.add_patch( mpl.patches.Circle( newPoint, radius = patchSize,
                                                  facecolor='#b7e8cc', edgecolor='black'  ))

              ax.text( newPoint[0], newPoint[1], speed, fontsize=15, horizontalalignment='center', verticalalignment='center

                 ax.set_title('Number of drones inserted: ' + str(len(run.drone_info)), fontdict={'fontsize':25})


             elif event.button == 3:
                 # Insert big red circles representing the source and target points

                 patchSize  = (xlim[1]-xlim[0])/50.0
                 if run.source is None:
                      run.source = (event.xdata, event.ydata)
                  ax.add_patch( mpl.patches.Circle( run.source, radius = patchSize, facecolor= '#ffd9d6', edgecolor='black',
                  ax.text( run.source[0], run.source[1], 'S', fontsize=15, horizontalalignment='center', verticalalignment='c

                 elif run.target is None:
                      run.target = (event.xdata, event.ydata)
                  ax.add_patch( mpl.patches.Circle( run.target, radius = patchSize, facecolor= '#ffd9d6', edgecolor='black',
                  ax.text( run.target[0], run.target[1], 'T', fontsize=15, horizontalalignment='center', verticalalignment='c

                 else:
                       print Fore.RED, "Source and Target already set", Style.RESET_ALL


             # Clear polygon patches and set up last minute \verb|ax| tweaks
             clearAxPolygonPatches(ax)
             applyAxCorrection(ax)
             fig.canvas.draw()


      return _enterPoints

    # 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']:
                    # Select algorithm to execute
                    algo_str = raw_input(Fore.YELLOW                                             +\
                            "Enter algorithm to be used to compute the tour:\n Options are:\n"   +\
                            " (odw)     One Dimensional Wavefront \n"                            +\
                            Style.RESET_ALL)

                    algo_str = algo_str.lstrip()

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

                    if   algo_str == 'odw':
                          tour = run.get_tour( algo_odw )
                    else:
                          print "Unknown option. No horsefly for you! ;-D "
                          sys.exit()

                    applyAxCorrection(ax)
                    fig.canvas.draw()

               elif event.key in ['c', 'C']:
                    # Clear canvas and states of all objects
                    run.clearAllStates()
                    ax.cla()

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

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

           return _keyPressHandler

    # Set up interactive canvas
    fig, ax =  plt.subplots()
    run = Single_PHO_Input()

    from matplotlib import rc

    # specify the custom font to use
    plt.rcParams['font.family'] = 'sans-serif'
    plt.rcParams['font.sans-serif'] = 'Times New Roman'

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

   ax.set_title("Enter drone positions, source and target onto canvas. \n (Enter speeds into the terminal, after inserting a d

    mouseClick   = 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 15.
Defines: `single_pho_run_handler` 15.

Uses: `algo_odw` 7c, `Single_PHO_Input` 6.

Chapter Index of Fragments

⟨PHO Algorithms 7bc, 9⟩ Referenced in 15.
⟨PHO Data Structures 6, 7a⟩ Referenced in 15.
⟨PHO Run Handlers 10⟩ Referenced in 15.

Chapter Index of Identifiers

`algo_odw`: 7c, 10.
`algo_pho_exact_given_order_of_drones`: 7c, 9.
`Single_PHO_Input`: 6, 10.
`single_pho_run_handler`: 10, 15.

Bibliography

[1] "France has started the drone mail carrier | earth chronicles news." http://earth-chronicles.com/science/france-has-started-the-drone-mail-carrier.html. (Accessed on 09/01/2019). 3

[2] J. S. Mitchell and C. H. Papadimitriou, "The weighted region problem: finding shortest paths through a weighted planar subdivision," *Journal of the ACM (JACM)*, vol. 38, no. 1, pp. 18–73, 1991. 3, 5

[3] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984. 4

[4] P. Briggs, J. D. Ramsdell, and M. W. Mengel, "Nuweb version 1.0 b1: A simple literate programming tool," 2001. 4

[5] J. S. Mitchell, "Shortest paths among obstacles in the plane," *International Journal of Computational Geometry & Applications*, vol. 6, no. 03, pp. 309–332, 1996. 5

[6] S. Diamond and S. Boyd, "Cvxpy: A python-embedded modeling language for convex optimization," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 2909–2913, 2016. 6

[7] E. L. Lawler, J. K. Lenstra, A. H. Rinnooy Kan, and D. B. Shmoys, "The traveling salesman problem; a guided tour of combinatorial optimization," 1985.

Appendices

Appendix A

Supporting Code

Main File

"../src/pho-main.py" 15≡

```
# Relevant imports for Package Handoff

from colorama import Fore, Style
from matplotlib import rc
from scipy.optimize import minimize
from sklearn.cluster import KMeans
import argparse
import inspect
import itertools
import logging
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

⟨ PHO Data Structures 6, ... ⟩
⟨ PHO Algorithms 7b, ... ⟩
⟨ PHO Run Handlers 10 ⟩

# Set up logging information relevant to this module
logger=logging.getLogger(__name__)
logging.basicConfig(level=logging.DEBUG)

def debug(msg):
    frame,filename,line_number,function_name,lines,index=inspect.getouterframes(
        inspect.currentframe())[1]
    line=lines[0]
    indentation_level=line.find(line.lstrip())
    logger.debug('{i} [{m}]'.format(
        i='.'*indentation_level, m=msg))


def info(msg):
    frame,filename,line_number,function_name,lines,index=inspect.getouterframes(
        inspect.currentframe())[1]
    line=lines[0]
    indentation_level=line.find(line.lstrip())
    logger.info('{i} [{m}]'.format(
        i='.'*indentation_level, m=msg))
```

```
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)

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()
    ax.lines[:]=[]
    applyAxCorrection(ax)

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()
    ax.lines[:]=[]
    applyAxCorrection(ax)


def animate_tour (sites, phi, horse_trajectories, fly_trajectories,
                  animation_file_name_prefix, algo_name,  render_trajectory_trails_p = False):
    """ This function can handle the animation of multiple
    horses and flies even when the the fly trajectories are all squiggly
    and if the flies have to wait at the end of their trajectories.

    A fly trajectory should only be a list of points! The sites are always the
    first points on the trajectories. Any waiting for the flies, is assumed to be
    at the end of their trajectories where it waits for the horse to come
    and pick them up.

    Every point on the horse trajectory stores a list of indices of the flies
    collected at the end point. (The first point just stores the dummy value None).
    Usually these index lists will be size 1, but there may be heuristics where you
    might want to collect a bunch of them together since they may already be waiting
    there at the pick up point.

    For each drone collected, a yellow circle is placed on top of it, so that
    it is marked as collected to be able to see the progress of the visualization
    as it goes on.

    """
    import numpy as np
    import matplotlib.animation as animation
    from   matplotlib.patches import Circle
    import matplotlib.pyplot as plt

    # Set up configurations and parameters for all necessary graphics
    plt.rc('text', usetex=True)
    plt.rc('font', family='serif')

    fig, ax = plt.subplots()
    ax.set_xlim([0,1])
    ax.set_ylim([0,1])
    ax.set_aspect('equal')

    ax.set_xticks(np.arange(0, 1, 0.1))
    ax.set_yticks(np.arange(0, 1, 0.1))

    # Turn on the minor TICKS, which are required for the minor GRID
    ax.minorticks_on()

    # customize the major grid
    ax.grid(which='major', linestyle='--', linewidth='0.3', color='red')

    # Customize the minor grid
    ax.grid(which='minor', linestyle=':', linewidth='0.3', color='black')

    ax.get_xaxis().set_ticklabels([])
    ax.get_yaxis().set_ticklabels([])

    mspan, _ = makespan(horse_trajectories)
    ax.set_title("Algo: " + algo_name + "  Makespan: " + '%.4f' % mspan , fontsize=25)

    number_of_flies  = len(fly_trajectories)
    number_of_horses = len(horse_trajectories)
    colors           = utils_graphics.get_colors(number_of_horses, lightness=0.5)

    ax.set_xlabel( "Number of drones: " + str(number_of_flies) + "\n" + r"$\varphi=$ " + str(phi), fontsize=25)
    ims                = []

    # Constant for discretizing each segment inside the trajectories of the horses
    # and flies.
    NUM_SUB_LEGS              = 2 # Number of subsegments within each segment of every trajectory

    # Arrays keeping track of the states of the horses
    horses_reached_endpt_p    = [False for i in range(number_of_horses)]
    horses_traj_num_legs     = [len(traj)-1 for traj in horse_trajectories] # the -1 is because the initial position of the hor
    horses_current_leg_idx    = [0 for i in range(number_of_horses)]
    horses_current_subleg_idx = [0 for i in range(number_of_horses)]
    horses_current_posn       = [traj[0]['coords'] for traj in horse_trajectories]

    # List of arrays keeping track of the flies collected by the horses at any given point in time,
    fly_idxs_collected_so_far = [[] for i in range(number_of_horses)]

    # Arrays keeping track of the states of the flies
    flies_reached_endpt_p    = [False for i in range(number_of_flies)]
    flies_traj_num_legs      = [len(traj)-1 for traj in fly_trajectories]
    flies_current_leg_idx    = [0 for i in range(number_of_flies)]
    flies_current_subleg_idx = [0 for i in range(number_of_flies)]
    flies_current_posn       = [traj[0] for traj in fly_trajectories]

    # The drone collection process ends, when all the flies AND horses
    # have reached their ends. Some heuristics, might involve the flies
    # or the horses waiting at the endpoints of their respective trajectories.
    image_frame_counter = 0
    while not(all(horses_reached_endpt_p + flies_reached_endpt_p)):

        # Update the states of all the horses
        for hidx in range(number_of_horses):
            if horses_reached_endpt_p[hidx] == False:
                htraj                       = [elt['coords'] for elt in horse_trajectories[hidx]]
             all_flys_collected_by_horse = [i          for elt in horse_trajectories[hidx] for i in elt['fly_idxs_picked_up']

                if horses_current_subleg_idx[hidx] <= NUM_SUB_LEGS-2:

                    horses_current_subleg_idx[hidx] += 1     # subleg idx changes
                    legidx    = horses_current_leg_idx[hidx] # the legidx remains the same

                  sublegidx = horses_current_subleg_idx[hidx] # shorthand for easier reference in the next two lines
```

```
                xcurr = np.linspace( htraj[legidx][0], htraj[legidx+1][0], NUM_SUB_LEGS+1 )[sublegidx]
                ycurr = np.linspace( htraj[legidx][1], htraj[legidx+1][1], NUM_SUB_LEGS+1 )[sublegidx]
                horses_current_posn[hidx]  = [xcurr, ycurr]


            else:
                horses_current_subleg_idx[hidx] = 0 # reset to 0
                horses_current_leg_idx[hidx]   += 1 # you have passed onto the next leg
                legidx    = horses_current_leg_idx[hidx]

            xcurr, ycurr = htraj[legidx][0], htraj[legidx][1] # current position is the zeroth point on the next leg
                horses_current_posn[hidx]  = [xcurr , ycurr]

                if horses_current_leg_idx[hidx] == horses_traj_num_legs[hidx]:
                    horses_reached_endpt_p[hidx] = True

            ####################......for marking in yellow during rendering
            fly_idxs_collected_so_far[hidx].extend(horse_trajectories[hidx][legidx]['fly_idxs_picked_up'])
         fly_idxs_collected_so_far[hidx] = list(set(fly_idxs_collected_so_far[hidx])) ### critical line, to remove duplic

    # Update the states of all the flies
    for fidx in range(number_of_flies):
        if flies_reached_endpt_p[fidx] == False:
            ftraj  = fly_trajectories[fidx]

            if flies_current_subleg_idx[fidx] <= NUM_SUB_LEGS-2:

                flies_current_subleg_idx[fidx] += 1
                legidx    = flies_current_leg_idx[fidx]

                sublegidx = flies_current_subleg_idx[fidx]
                xcurr = np.linspace( ftraj[legidx][0], ftraj[legidx+1][0], NUM_SUB_LEGS+1 )[sublegidx]
                ycurr = np.linspace( ftraj[legidx][1], ftraj[legidx+1][1], NUM_SUB_LEGS+1 )[sublegidx]
                flies_current_posn[fidx]  = [xcurr, ycurr]

            else:
                flies_current_subleg_idx[fidx] = 0 # reset to zero
                flies_current_leg_idx[fidx]   += 1 # you have passed onto the next leg
                legidx    = flies_current_leg_idx[fidx]

            xcurr, ycurr = ftraj[legidx][0], ftraj[legidx][1] # current position is the zeroth point on the next leg
                flies_current_posn[fidx]  = [xcurr , ycurr]

                if flies_current_leg_idx[fidx] == flies_traj_num_legs[fidx]:
                    flies_reached_endpt_p[fidx] = True

    objs = []
    # Render all the horse trajectories uptil this point in time.
    for hidx in range(number_of_horses):
        traj               = [elt['coords'] for elt in horse_trajectories[hidx]]
        current_horse_posn = horses_current_posn[hidx]

        if horses_current_leg_idx[hidx] != horses_traj_num_legs[hidx]: # the horse is still moving

              xhs = [traj[k][0] for k in range(1+horses_current_leg_idx[hidx])] + [current_horse_posn[0]]
              yhs = [traj[k][1] for k in range(1+horses_current_leg_idx[hidx])] + [current_horse_posn[1]]

        else: # The horse has stopped moving
              xhs = [x for (x,y) in traj]
              yhs = [y for (x,y) in traj]

        horseline, = ax.plot(xhs,yhs,'-',linewidth=5.0, markersize=6, alpha=0.80, color='#D13131')
      horseloc   = Circle((current_horse_posn[0], current_horse_posn[1]), 0.015, facecolor = '#D13131', edgecolor='k',  al
                horsepatch = ax.add_patch(horseloc)
                objs.append(horseline)
                objs.append(horsepatch)

            # Render all fly trajectories uptil this point in time
            for fidx in range(number_of_flies):
                traj               = fly_trajectories[fidx]
                current_fly_posn   = flies_current_posn[fidx]

                if flies_current_leg_idx[fidx] != flies_traj_num_legs[fidx]: # the fly is still moving

                      xfs = [traj[k][0] for k in range(1+flies_current_leg_idx[fidx])] + [current_fly_posn[0]]
                      yfs = [traj[k][1] for k in range(1+flies_current_leg_idx[fidx])] + [current_fly_posn[1]]

                else: # The fly has stopped moving
                      xfs = [x for (x,y) in traj]
                      yfs = [y for (x,y) in traj]

                if render_trajectory_trails_p:
                    flyline, = ax.plot(xfs,yfs,'-',linewidth=2.5, markersize=6, alpha=0.32, color='b')
                    objs.append(flyline)


                # If the current fly is in the list of flies already collected by some horse,
                # mark this fly with yellow. If it hasn't been serviced yet, mark it with blue
                service_status_col = 'b'
                for hidx in range(number_of_horses):
                    #print fly_idxs_collected_so_far[hidx]
                    if fidx in fly_idxs_collected_so_far[hidx]:
                        service_status_col = 'y'
                        break

                flyloc   = Circle((current_fly_posn[0], current_fly_posn[1]), 0.013,
                                  facecolor = service_status_col, edgecolor='k', alpha=1.00)
                flypatch = ax.add_patch(flyloc)
                objs.append(flypatch)

            print ".......................”
            print "Appending to ims ", image_frame_counter
            ims.append(objs)
            image_frame_counter += 1

        from colorama import Back

        debug(Fore.BLACK + Back.WHITE + "\nStarted constructing ani object"+ Style.RESET_ALL)
        ani = animation.ArtistAnimation(fig, ims, interval=70, blit=True, repeat=True, repeat_delay=500)
        debug(Fore.BLACK + Back.WHITE + "\nFinished constructing ani object"+ Style.RESET_ALL)

        debug(Fore.MAGENTA + "\nStarted writing animation to disk"+ Style.RESET_ALL)
        #ani.save(animation_file_name_prefix+'.avi', dpi=100)
        #ani.save('reverse_horsefly.avi', dpi=300)
        debug(Fore.MAGENTA + "\nFinished writing animation to disk"+ Style.RESET_ALL)

        plt.show()

    if __name__=="__main__":
         print "Running Package Handoff"
         single_pho_run_handler()

    ◇
```

Uses: `single_pho_run_handler` 10.

Algorithmic Utilities

"../src/utils_algo.py" 20≡

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

def vector_chain_from_point_list(pts):
    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

def length_polygonal_chain(pts):
    vec_chain = vector_chain_from_point_list(pts)

    acc = 0
    for vec in vec_chain:
        acc = acc + np.linalg.norm(vec)
    return acc
def pointify_vector (x):
    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):
       return [item for sublist in l for item in sublist]
def print_list(xs):
    for x in xs:
        print x
def partial_sums( xs ):
    psum = 0
    acc = []
    for x in xs:
        psum = psum+x
        acc.append( psum )
    return acc
def are_site_orderings_equal(sites1, sites2):

    for (x1,y1), (x2,y2) in zip(sites1, sites2):
        if (x1-x2)**2 + (y1-y2)**2 > 1e-8:
            return False
    return True
def bunch_of_non_uniform_random_points(numpts):
    cluster_size = int(np.sqrt(numpts))
    numcenters   = cluster_size

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

    scale, points = 4.0, []
    for c in centers:
        cx, cy = c[0], c[1]
       # For current center $c$ of this loop, generate \verb|cluster_size| points uniformly in a square centered at it

        sq_size      = min(cx,1-cx,cy, 1-cy)
        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))


    # Whatever number of points are left to be generated, generate them uniformly inside the unit-square

    num_remaining_pts = numpts - cluster_size * numcenters
    remaining_pts = scipy.rand(num_remaining_pts, 2).tolist()
    points.extend(remaining_pts)


    return points

def write_to_yaml_file(data, dir_name, file_name):
   import yaml
   with open(dir_name + '/' + file_name, 'w') as outfile:
     yaml.dump( data, outfile, default_flow_style = False)
```

◇

Graphical Utilities

"../src/utils_graphics.py" 21≡

```
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]

# Borrowed from https://stackoverflow.com/a/9701141
import numpy as np
import colorsys

def get_colors(num_colors, lightness=0.2):
    colors=[]
    for i in np.arange(60., 360., 300. / num_colors):
        hue        = i/360.0
        saturation = 0.95
        colors.append(colorsys.hls_to_rgb(hue, lightness, saturation))
    return colors
```

◇