Analyses of Experimental Heuristics for Package-Handoff Type Problems

Gaurish Telang

Contents

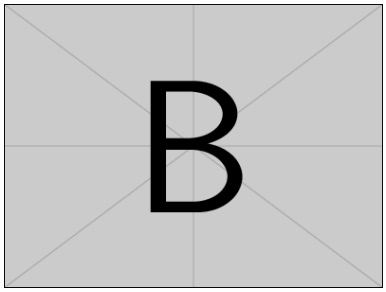
Chapter 1

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.
- *For the single delivery package handoff problem, a slower drone, always hands off the package to a faster drone, in any optimal schedule.*
- *All drones involved in the optimal schedule start moving at time $t = 0$. The drone not involved can remain stationary since they don't participate in the package transport.*
- *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.*
- *The path of the package is a monotonic polygonal curve with respect to the direction $\overrightarrow{ST}$ no matter what the intial positions P_i or speeds u_i of the drones.*

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. [1]

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.
- $(S = H_{i_0}), H_{i_1} \ldots H_{i_k}$ for $0 \leq i_0, \ldots i_k \leq n$ stand for points where the drones with labels $i_0, \ldots i_k$ handle the package in that oder. More precisely H_{i_j} is the point where drone i_{j-1} hands off the package to drone i_j for $1 \leq j \leq k$.

Propagating 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

[1] 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.

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.

Algorithm 1: How to write algorithms

Data: (P_i, u_i) for $1 \leq i \leq n$ where $P_i \in \mathbb{R}^2$ and $u_i \geq 0$, $S, T \in \mathbb{R}^2$
Result: A polygonal path $(S = H_{i_0}), H_{i_1} \dots H_{i_k}, T$ representing the path travelled by the package.
initialization;
while *not at end of this document* **do**
    read current;
    if *understand* **then**
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end

Chapter Index of Fragments

None.

Chapter Index of Identifiers

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

Appendices

Appendix A

Supporting Code

The following file, contains some common algorithmic utilities

"../src/utils_algo.py" 10≡

```
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)


    ◇
```

"../src/utils_graphics.py" 11≡

```
    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)

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 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 a site

                 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})


             elif event.button == 3:
                 # Insert big red circle representing initial position of horse and fly

                 inithorseposn     = (event.xdata, event.ydata)
                 run.inithorseposn = inithorseposn
                 patchSize         = (xlim[1]-xlim[0])/100.0

                 ax.add_patch( mpl.patches.Circle( inithorseposn,radius = patchSize,
                                                   facecolor= '#D13131', edgecolor='black' ))


             # Clear polygon patches and set up last minute \verb|ax| tweaks

             clearAxPolygonPatches(ax)
             applyAxCorrection(ax)
             fig.canvas.draw()


    return _enterPoints

# 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
```

◇