Coordinating Heterogenous Agents for Fast Package Delivery — The Package Handoff Problem

Jie Gao [1], Kien Huynh [1], Joseph S.B. Mitchell [2], Gaurish Telang[2]

[1] Department of Computer Science, Stony Brook University
[2] Department of Applied Mathematics and Statistics, Stony Brook University

October 14, 2019

Abstract

How do you get a package from an initial location S to a destination point T using a fleet of "heterogenous" carrier agents (e.g. drones, taxis). By "heterogenous" we mean the two agents can have different capabilities like different maximum speed or different fuel capacity.

In the simplest version of the category of problems, we are given as input the initial locations of n agents in $\mathbb{R}^2$ each capable of a maximum speed $u_i > 0$ (where u_i need *not* be equal to v_j for $i \neq j$). Each agent can pick up the package and move to another point to *rendezvous with* and *hand off* the package to another agent. This other agent then either proceeds to T or decides to meet with and hand off the package to another agent, until the last agent decides to head directly to T.

The objective is to get the agents to cooperate to send the package from S to T in the least possible time. We call this the *Package Handoff Problem*.

To solve this problem and its various avatars we need to

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

This report is an algorithmic study of various heuristics developed to solve different variants of Package Handoff.

Contents

Chapter 1

Single Package Handoff

1.1 Introduction

How do you get a package from an initial location S to a destination point T using a fleet of "heterogenous" carrier agents (e.g. drones, taxis). By "heterogenous" we mean the two agents can have different capabilities like different maximum speed or different fuel capacity.

In the simplest version of the category of problems, we are given as input the initial locations of n agents in $\mathbb{R}^2$ each capable of a maximum speed $u_i > 0$ (where u_i need not be equal to v_j for $i \neq j$). Each agent can pick up the package and move to another point to *rendezvous with* and *hand off* the package to another agent. This other agent then either proceeds to T or decides to meet with and hand off the package to another agent [1] and so on and so forth.

The objective is to get the agents to cooperate to send the package from S to T in the least possible time. We call this the *Package Handoff Problem*.

To solve this problem and its various avatars [2] we need to

1. Figure out which subset $S = \{i_1, i_2, \ldots i_k\}$ of the drones are used in the optimal schedule.

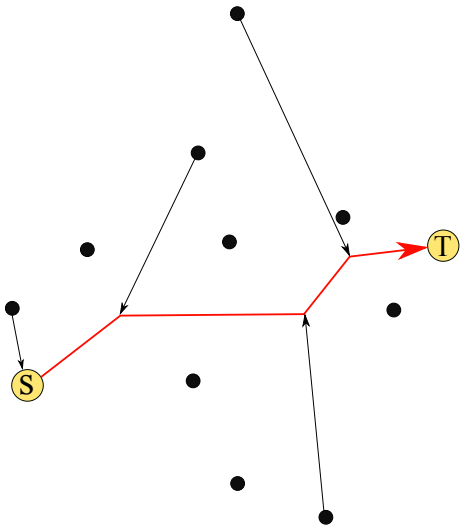

Figure 1.1: An instance of the Package Handoff problem for a single package being transported from S to T. Agents are located at the dots marked in black. The package travels along the red path. The agents all have different velocities, and in this example, assumed to have infinite battery capacity.

[1] If it makes the package get to T faster

[2] Say when there are multiple packages to be delivered or a bound on fuel

2. Find the order in which the handoffs happend between the drones used in a schedule.
3. Calculate the "handoff" points where drone i_m hands over the package to drone i_{m+1} [3]

A real world instance of the basic Package Handoff problem, as described in the abstract, is when a ride hailing service must co-ordinate its fleet of taxis to transport a passenger from a given location in the quickest possible time to the target destination on the map. In this model, a passenger "hops rides" when two taxis meet: a taxi first gets to the passenger and takes him/her to a point where it rendezvous with another taxi, at which point the passenger swaps taxis. This process continues until the passenger hops onto a taxi that goes straight to the target.

This package handoff process is depicted in Figure 1.1.

1.2 A note on source code

Many of the heuristics algorithms described here are implemented as literate programs [1] in Python 2.7.12 using the NuWeb tool [2] available from `http://nuweb.sourceforge.net/` alongside associated theoretical and empirical analysis. All the algorithmic code goes into the file `pholib.py`, and any associated helper codes go into the files which are named as `utils_*.py`. The code for these utility files has been given in the appendices.

The `pholib.py` file looks like

"src/pholib.py" 3≡

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

⟨ Algorithms 7 ⟩
⟨ Experiments ? ⟩
⟨ Run Handlers 23 ⟩
⟨ Plotting 27 ⟩
◇
```

[3] The last drone in the computed schedule, of course, flies directly to T

The chapters are devoted to fleshing out the chunks ⟨`Algorithms`⟩and ⟨`Experiments`⟩. The ⟨`Run Handlers`⟩and ⟨`Plotting`⟩ chunks is mainly to deal with interactive matplotlib input, and as such are boring and banished to the Appendix ☺.

All source code files are tangled to the `src` directory. The point of entry for the code are `main*.py` which are implemented separately in the `src` directory, since their contents can change based on what library code is being called for during development and testing. Since these files are very short and the mechanics clear, they are implemented as standalone files (i.e. not inside this document) but directly in the `src` folder itself. To run the code in interactive mode run the code as `python src/main_interactive.py` on a Unix / Windows terminal in the root folder of the project [4].

For a short overview of previous work on this problem see Appendix.

The `README` file containing instructions for running the source code and experiments is listed in the appendix (also available on the Github repository)

Each of the following sections correspond to a fixed variant of the package handoff problem and describe algorithms for that specific variant. Enough talk! Onto algorithms!

1.3 Unlimited Fuel, Different Drone Speeds

Much of the machinery developed in solving this basic basic basic question will be generalized and extended to other variants of the package handoff problem.

We repeat the problem definition and fix some notation that will be used for the remainder of the section

> *We are given as input the initial locations P_i of n agents in $\mathbb{R}^2$ each capable of a maximum speed $u_i > 0$ (where u_i need not be equal to u_j for $i \neq j$). Each agent can pick up the package and move to another point to rendezvous with and hand off the package to another agent. This other agent then either proceeds to T or decides to meet with and hand off the package to another agent and so on. The objective is to get the agents to cooperate to send the package from S to T in the least possible time.*

We represent the handoff points as follows $H_{i_1} \ldots H_{i_k}$ for $0 \leq i_0, \ldots i_k \leq n$ stand for points where the drones with labels $i_0, \ldots i_k$ hand the package off 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 $1 \leq j \leq k$.

A solution to the package handoff problem is completely specified by computing the handoff points *and* the drone ids involved in the exchange at each handoff point.

The optimal schedule is denoted OPT. It is easy to see the statements of the following structural lemma always hold for OPT.

[4] This code has been tested on an 64 bit machine running Linux Mint 18.3 (Sylvia) running the Linux Kernel version 4.10.0-38-generic with an Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz CPU

Lemma 1. *In OPT*

- ***A.*** *A package is always transferred to a faster drone at a handoff point.*
- ***B.*** *A drone handles a package at most once i.e. if a drone hands off the package, it will never be involved in a handling that package again.*
- ***C.*** *All drones involved in the handoff start moving simultaneously at time $t = 0$*
- ***D.*** *No two drones wait at a rendezvous point before rendezvous happens.* [a]
- ***E.*** *The path of the package is a radially monotone piecewise straight polygonal curve with respect to the direction ST no matter what the initial positions P_i or speeds u_i of the drones.*
- ***F.*** $\frac{|ST|}{v_{max}}$ *is a (trivial) lower bound for OPT, where v_{max} denotes the speed of the fastest drone.*

[a] waiting can happen in other problem variants say when there is limited fuel or only a finite set of allowed rendezvous points

Proof. TODO!

□

1.3.1 Handoff in a fixed order

If we know the drones involved in the handoff *along with* the order of handoff then we can compute the handoff points — and hence the path of the package — exactly via convex optimization as outlined in Lemma 2. This fact will be exploited in many heuristics: such methods will compute a subset of drones involved in the handoff (alongwith the handoff order) followed by a call to the convex program.

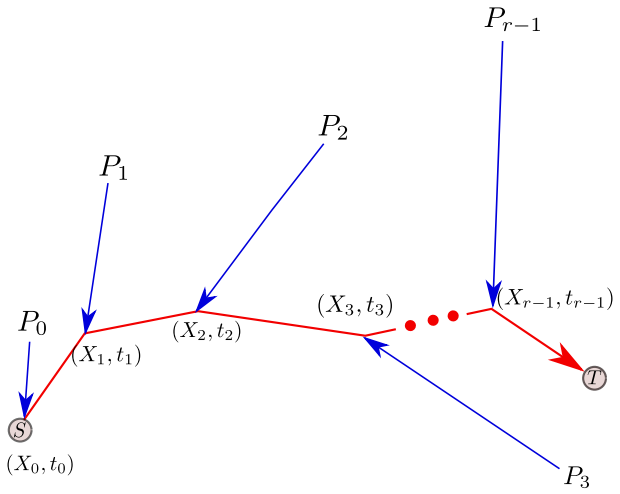

Figure 1.2: The path of the package is shown in red. The drones involved in the handoff are labelled P_i in the prespecified handoff order.

Lemma 2. *Given as input are drones with initial positions $P_i \in \mathbb{R}^2$, with speeds $u_i > 0$ for $1 \leq i \leq r-1$, the intial position S and final destination T for the package.* [a] *The drones are expected to transport the package by handing of the package in the order $1, 2, \ldots, r$. Let t_i denote the departure time on a global clock from the i'th handoff point X_i.*

Then the minimum time and handoff points for transporting the package and the handoff points can be calculated by the following convex program

$$\min_{t_i, X_i} \quad t_{r-1} + \frac{||T - X_{r-1}||}{u_{r-1}}$$

subject to the constraints

$$\begin{aligned} X_0 &= S \\ t_i &\geq \frac{||P_i - X_i||}{u_i} && 0 \leq i \leq r-1 \\ t_i + \frac{||X_{i+1} - X_i||}{u_i} &\leq t_{i+1} && 0 \leq i \leq r-2 \end{aligned}$$

[a] See Figure 1.3.1 for an illustration of the notation used in this lemma

Proof. TODO!

□

The following function is just an implentation of the convex program just described. Here `drone_info` is a list of tuples, where each tuple consits of the initial position and speed of the drone. The order of the drones is assumed to be that in which the list of drones is provided. `source` and `target` are just coordinate locations of S and T respectively. We use the CVXPY [3] library as a black-box convex optimization solver.

⟨ *Algorithms 7* ⟩ ≡

```
def algo_pho_exact_given_order_of_drones ( drone_info, source, target ):
    import cvxpy as cp

    source = np.asarray(source)
    target = np.asarray(target)

    r = len(drone_info)
    source = np.asarray(source)
    target = np.asarray(target)

    # Variables for rendezvous points of drone with package
    X, t = [], []
    for i in range(r):
       X.append(cp.Variable(2)) # vector variable
       t.append(cp.Variable( )) # scalar variable

    # Constraints
    constraints_S = [  X[0] == source ]

    constraints_I = []
    for i in range(r):
      constraints_I.append(0.0 <= t[i])
      constraints_I.append(t[i] >= cp.norm(np.asarray(drone_info[i][0])-X[i])/drone_info[i][1]

    constraints_L = []
    for i in range(r-1):
      constraints_L.append(t[i] + cp.norm(X[i+1] - X[i])/drone_info[i][1] <= t[i+1])

    objective = cp.Minimize(t[r-1]+cp.norm(target-X[r-1])/drone_info[r-1][1])

    prob = cp.Problem(objective, constraints_S + constraints_I + constraints_L)
    print Fore.CYAN
    prob.solve(solver=cp.SCS,verbose=True)
    print Style.RESET_ALL

    package_trail = [ np.asarray(X[i].value) for i in range(r) ] + [ target ]
    return package_trail
```

◇

Fragment referenced in 3.

We next describe a heuristic that use Continuous Dijkstra [4] type approach in computing approximate solutions to OPT.

1.3.2 One Dimensional Greedy Wavefront

In this heuristic we first constrain the package to travel along the line $\vec{ST}$, then compute the subset of the drones involved in the schedule, and finally pass of the list of drones involved to the convex program given in Lemma 2 to calculate the rendezvous points.

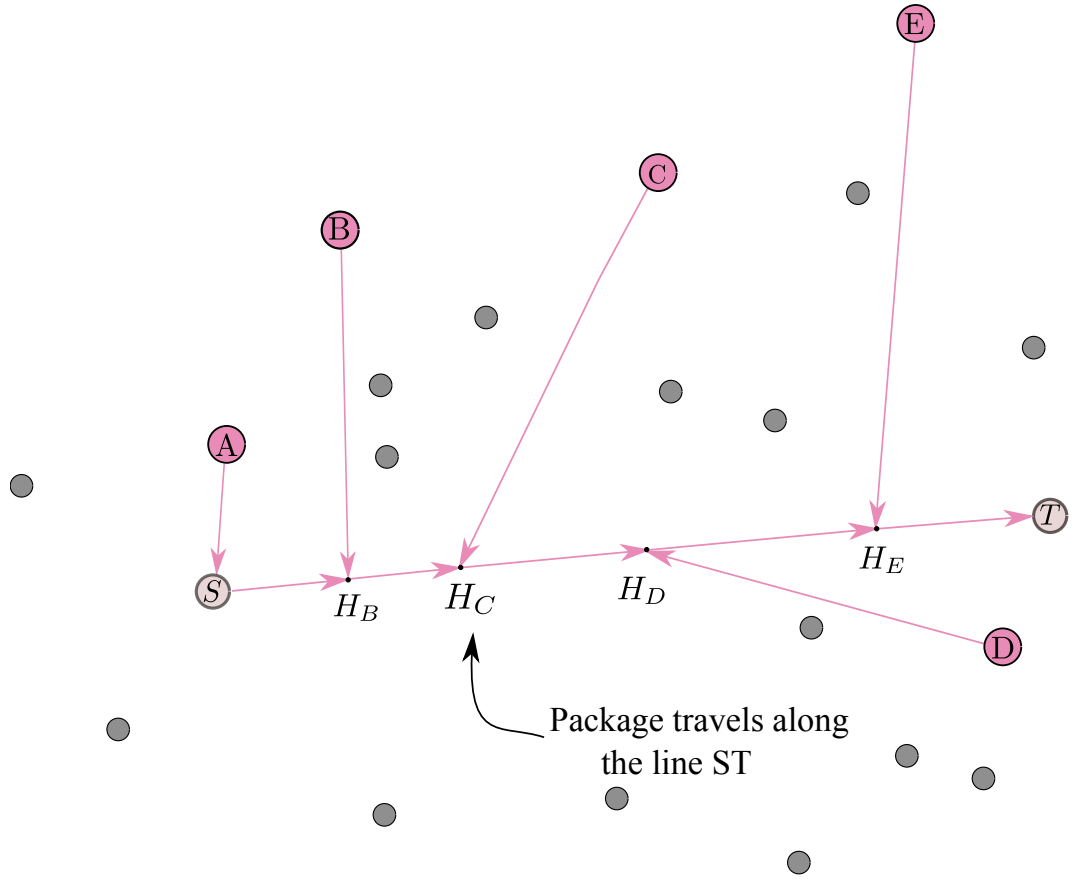

Figure 1.3: The package travels along the straight line $\vec{ST}$. The point where drone A hands off the package to drone B depicted as H_B, and similarly for other drones. Drones involved in the handoff are marked in pink. Those not involved are marked in gray. Two drones may have different speed.

Algorithm 1: One dimensional greedy wavefront

Input:

1. Coordinates of initial position of source S and target T of the package.
2. Coordinates of the onitial positions P_i of each drone $1 \leq i \leq n$.
3. Maximum possible speed u_i of each drone $1 \leq i \leq n$.

Output:

1. t^*: The time required for the package to get from S to T.
2. $L = (i_1, i_2, \ldots, i_k)$: An ordered list of indices of the drones involved in the handoff.
3. $\mathcal{H} = \{ S \} \cup \{ H_{i_j} \mid j \geq 2 \}$: An ordered list of handoff points. $H_{i_j} \in \mathbb{R}^2$ is where the drone with index i_j picks up the package *from* drone i_{j-1}.

$t \leftarrow 0$ // Time on the global clock
$wavelets \leftarrow [(i, 0) \mid 0 \leq i \leq n]$ // Active Wavelets: indices and current radius
/* Find first wavelet to reach S and update $wavelets$ */
/* Start wavelet expansion from S */
for $i \leftarrow 1$ ***to*** r **do**
    while $n \geq c_i$ **do**
        $C \leftarrow C \cup \{c_i\}$
        $n \leftarrow n - c_i$
    end
end
return $t^*, L, \mathcal{H}$

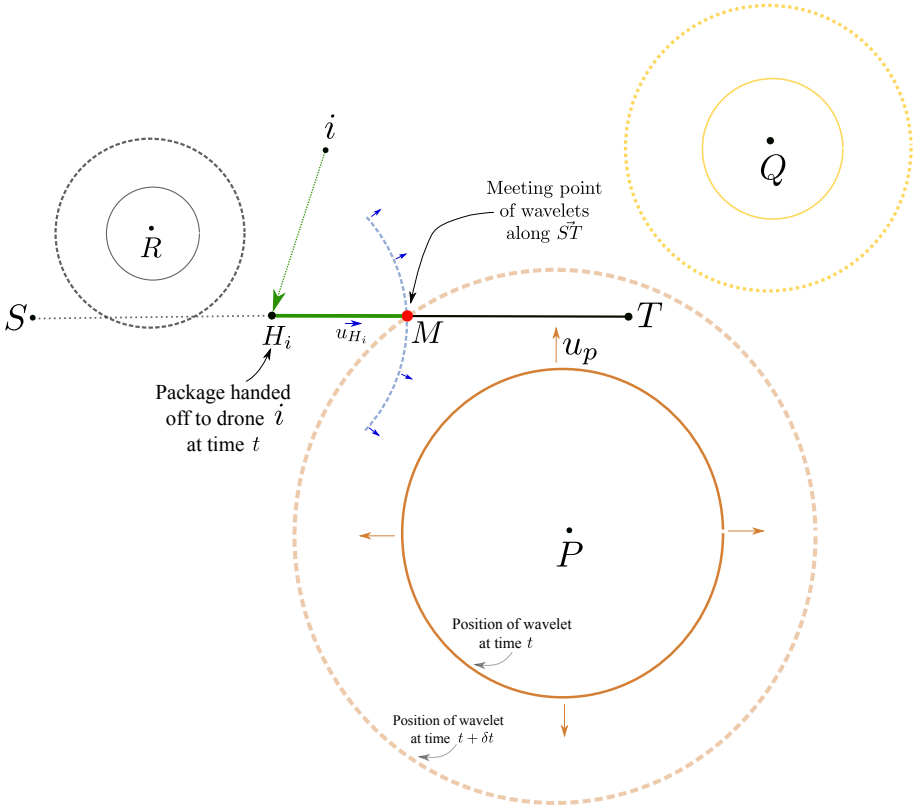

Figure 1.4: Intersection of two expanding wavelets along $\vec{H_iT}$. The figure shows snapshots of two times; one at time t when the package has just been handed off to drone i at H_i and another at time $t+\delta t$ when a wavelet corresponding a drone faster than drone i meets the wavelet expanding from H_i with speed u_i at M

Chapter 2

Multiple Package Handoff

2.1 Section 1

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2.2 Section 2

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2.3 Section 3

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Bibliography

[1] Donald E. Knuth. "Literate Programming". In: *Comput. J.* 27.2 (May 1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. URL: http://dx.doi.org/10.1093/comjnl/27.2.97.

[2] Preston Briggs. "Nuweb Version 0.87 b: A simple literate programming tool". In: *Published on the World-Wide Web by preston@ cs. rice. edu* (1992).

[3] Steven Diamond and Stephen Boyd. "CVXPY: A Python-Embedded Modeling Language for Convex Optimization". In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5.

[4] Joseph SB Mitchell et al. "Geometric shortest paths and network optimization". In: *Handbook of computational geometry* 334 (2000), pp. 633–702.

[5] Hang Ma et al. "Multi-Agent Path Finding with Payload Transfers and the Package-Exchange Robot-Routing Problem". In: (2016).

[6] Tsz-Chiu Au et al. "Multirobot systems". In: *IEEE Intelligent Systems* 6 (2017), pp. 3–5.

[7] Hang Ma and Sven Koenig. "Optimal target assignment and path finding for teams of agents". In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2016, pp. 1144–1152.

[8] Ariel Felner et al. "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges". In: *Tenth Annual Symposium on Combinatorial Search*. 2017.

[9] *"ORPHANS PREFERRED - Wanted: Young, skinny, wiry fellows not over eighteen. Must be expert riders, willing to risk death daily." Pony Express ad, 1860 : vintageads*. https://www.reddit.com/r/vintageads/comments/brxqiy/orphans_preferred_wanted_young_skinny_wiry/. (Accessed on 10/04/2019).

[10] *Pony Express - Wikipedia*. https://en.wikipedia.org/wiki/Pony_Express. (Accessed on 10/04/2019).

[11] *PonyExpressRoute.jpg (586×361)*. https://www.legendsofamerica.com/wp-content/uploads/2018/05/PonyExpressRoute.jpg. (Accessed on 10/04/2019).

[12] *Zipline*. https://flyzipline.com/impact/. (Accessed on 10/04/2019).

[13] *How we're using drones to deliver blood and save lives | Keller Rinaudo - YouTube*. https://www.youtube.com/watch?v=73rUjrow5pI. (Accessed on 10/04/2019).

[14] *Matternet*. https://mttr.net/company. (Accessed on 10/04/2019).

[15] *Mumbai's amazing dabbawalas*. [Online; accessed 9. Oct. 2019]. 2019. URL: http://specials.rediff.com/money/2005/nov/11spec.htm.

[16] Works That Work Magazine. *Dabbawallas: Delivering Excellence by Meena Kadri (Works That Work magazine)*. [Online; accessed 9. Oct. 2019]. 2019. URL: https://worksthatwork.com/1/dabbawallas.

Appendices

Appendix A

History and Previous Work

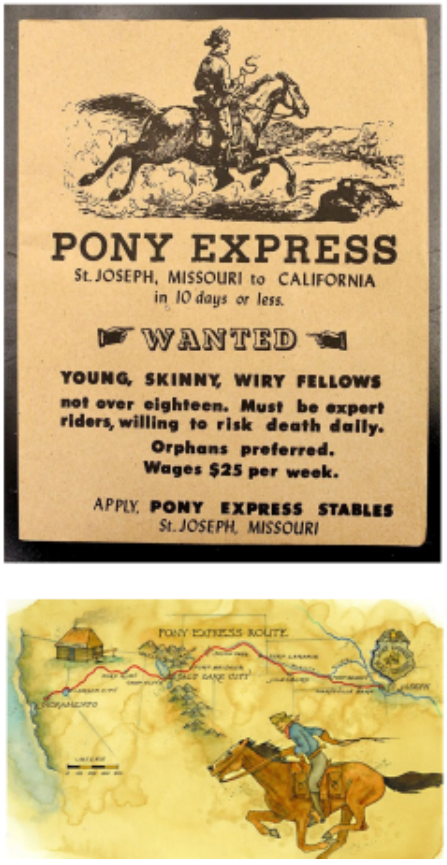

Figure A.1: A job application poster and a relay route used for the Pony Express. Images taken from [9] and [11] respectively.

A system of using relays for delivering packages is not a particularly new idea. A famous (and short-lived!) example of such a relay system was the Pony Express company which was used a system of a relay of horse riders tp transport mail from St. Joseph, Missouri to Sacramento, California.

To quote the Wikipedia article

> *"Operated by Central Overland California and Pike's Peak Express Company, the Pony Express was a great financial investment to the U.S. During its 18 months of operation, it reduced the time for messages to travel between the Atlantic and Pacific coasts to about 10 days. It became the West's most direct means of east-west communication before the transcontinental telegraph was established (October 24, 1861), and was vital for tying the new U.S. state of California with the rest of the United States. The Pony Express demonstrated that a unified transcontinental system of communications could be established and operated year-round. "*

While the invention of the telegraph might have run the Pony Express out of business, the idea of using relay agents such as drones — instead of horses! — to transfer packages can have applications today for sending physical goods (which of course can't be telegraphed! ☺) such as life-saving medicinces in

under-developed countries or in disaster relief areas. ZipLine[12] [13] and Matternet [14] are just two of the companies which are involved in building networks of drones for precisely such missions.

Figure A.2: Dabbawallas exchanging lunchboxes (dabbas) at a relay point. Image from [16]

Another relay system for package deliveries (135 years old and still functioning!) is that of the *dabbawallas* [1] used for transporting lunch boxes from homes and restaurants to people at work in Mumbai, India. To quote from [15]

> *Four thousand five hundred semi-literate dabbawalas collect and deliver 175,000 packages within hours. What should we learn from this unique, simple and highly efficient 120-year-old logistics system? [...] After the customer leaves for work, her lunch is packed into a tiffin provided by the dabbawala. A color-coded notation on the handle identifies its owner and destination. Once the dabbawala has picked up the tiffin, he moves fast using a combination of bicycles, trains and his two feet.*
>
> *A BBC crew filming dabbawalas in action was amazed at their speed. "Following our dabbawala wasn't easy, our film crew quickly lost him in the congestion of the train station. At Victoria Terminus we found other fast moving dabbawalas, but not our subject... and at Mr Bhapat's ayurvedic pharmacy, the lunch had arrived long before the film crew," the documentary noted wryly. So, how do they work so efficiently?*
>
> *The entire system depends on teamwork and meticulous timing. Tiffins are collected from homes between 7.00 am and 9.00 am, and taken to the nearest railway station. At various intermediary stations, they are hauled onto platforms and sorted out for area-wise distribution, so that a single tiffin could change hands three to four times in the course of its daily journey.*
>
> *At Mumbai's downtown stations, the last link in the chain, a final relay of dabbawalas fan out to the tiffins' destined bellies. Lunch hour over, the whole process moves into reverse and the*

[1] literally: lunchbox carriers

tiffins return to suburban homes by 6.00 pm.

See https://youtu.be/dX-0el2wuEU for a short video on the dabbawallas.

Appendix B

README

This README file can be read more clearly alongside its appropriate formatting at https://github.com/gtelang/packagehandoff

"README.md" 16≡

```
To run this code, you will need a distribution of Python 2.7.12 along
with the following libraries

- networkx
- matplotlib
- numpy
- scipy
- cgal-bindings
- cvxpy

All source code is contained in the .web file. If you modify the file, the
resulting code and corresponding description file can be weaved and tangled
with the script ‘weave-tangle.sh‘. You will need the

- [pdflatex](https://linux.die.net/man/1/pdflatex)
- [nuweb](http://nuweb.sourceforge.net/)
- [asymptote](http://asymptote.sourceforge.net/)

executables to be somewhere on your system’s path.

All source code is in the ‘src’ directory. The asy2d,asy3d and docs
folders can be neglected since they contain images and documents
referenced in the ‘packagehandoff.pdf’ file. From the point of view of
_running_ the code they can be ignored.
```

◇

Appendix C

utils_graphics.py

This file contains useful functions for visualization and plotting functions described in the previous chapters.

"src/utils_graphics.py" 18≡

```
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
```

◇

Appendix D

utils_algo.py

This file contains useful functions for writing algorithms described in the previous chapters.

"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 i

        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=(cl
        loc_pts_y    = np.random.uniform(low=cy-sq_size/scale, high=cy+sq_size/scale, size=(cl
        points.extend(zip(loc_pts_x, loc_pts_y))


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

    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)
```

◇

Appendix E

Implementation of ⟨**Run Handlers**⟩

This chunk contains code required for the interactive input of sites and agents onto the canvas.

⟨ *Run Handlers* 23 ⟩ ≡

```
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, animate_tour_p=False, plot_tour_p=False):
           return algo( self.drone_info,
                        self.source,
                        self.target,
                        animate_tour_p,
                        plot_tour_p    )

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

def single_pho_run_handler():
    import random
    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    = np.random.uniform() # float(raw_input('What speed do you want for
              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], "{:.2f}".format(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='center' )

              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='center' )
              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 clea
                clearAxPolygonPatches(ax)
                if   algo_str == 'odw':
                      tour = run.get_tour( algo_odw, plot_tour_p=True )
                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()

                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 drone at a particular position)")

    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 3.

Appendix F

Implementation of ⟨**Plotting**⟩

We typically plot the tours onto a separate window if the boolean switch `plot_tour_p` is set to `True` while calling the algorithm. The path of the package is shown in bold red. The paths of the drones from their initial positions to the point where they pick up the package from another drone are shown in blue.

An example output from the `plot_tour` function is shown below.

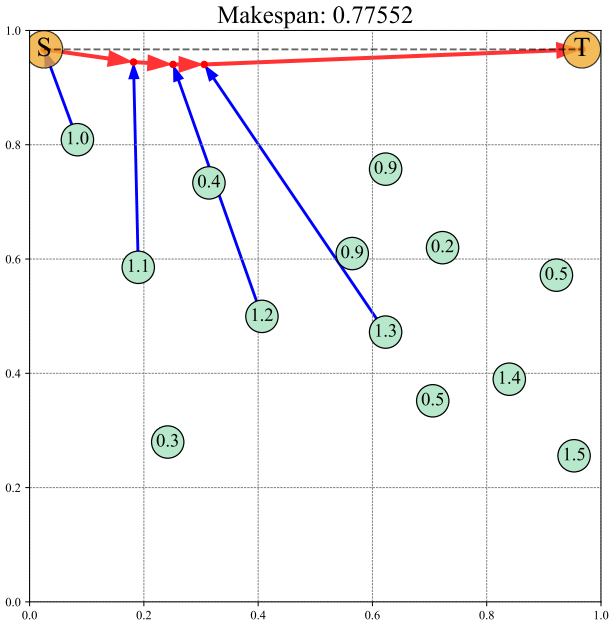

Figure F.1: The numbers inside the circle indicate the speed of the drone

⟨ *Plotting* 27 ⟩ ≡

```
def plot_tour(fig, ax, figtitle, source, target,
              drone_info, used_drones, package_trail,
              xlims=[0,1],
              ylims=[0,1],
```

```
          aspect_ratio=1.0,
          speedfontsize=4,
          speedmarkersize=10,
          sourcetargetmarkerfontsize=4,
          sourcetargetmarkersize=10 ):

import matplotlib.ticker as ticker
ax.set_aspect(aspect_ratio)
ax.set_xlim(xlims)
ax.set_ylim(ylims)

plt.rc('font', family='serif')

# Draw the package trail
xs, ys = extract_coordinates(package_trail)
ax.plot(xs,ys, 'ro', markersize=5 )
for idx in range(len(xs)-1):
      plt.arrow( xs[idx], ys[idx], xs[idx+1]-xs[idx], ys[idx+1]-ys[idx],
                **{'length_includes_head': True,
                   'width': 0.007 ,
                   'head_width':0.01,
                   'fc': 'r',
                   'ec': 'none',
                   'alpha': 0.8})


# Draw the source, target, and initial positions of the robots as bold dots
xs,ys = extract_coordinates([source, target])
ax.plot(xs,ys, 'o', markersize=sourcetargetmarkersize, alpha=0.8, ms=10, mec='k', mfc='#F1
#ax.plot(xs,ys, 'k--', alpha=0.6 ) # light line connecting source and target

ax.text(source[0], source[1], 'S', fontsize=sourcetargetmarkerfontsize,\
        horizontalalignment='center',verticalalignment='center')
ax.text(target[0], target[1], 'T', fontsize=sourcetargetmarkerfontsize,\
        horizontalalignment='center',verticalalignment='center')

xs, ys = extract_coordinates( [ drone_info[idx][0] for idx in range(len(drone_info)) ]  )
ax.plot(xs,ys, 'o', markersize=speedmarkersize, alpha = 0.5, mec='None', mfc='#b7e8cc' )

# Draw speed labels
for idx in range(len(drone_info)):
     ax.text( drone_info[idx][0][0], drone_info[idx][0][1], format(drone_info[idx][1],'.3f
              fontsize=speedfontsize, horizontalalignment='center', verticalalignment='cen

# Draw drone path from initial position to interception point
for pt, idx in zip(package_trail, used_drones):
     initdroneposn = drone_info[idx][0]
     handoffpoint  = pt
```

```
        xs, ys = extract_coordinates([initdroneposn, handoffpoint])
        plt.arrow( xs[0], ys[0], xs[1]-xs[0], ys[1]-ys[0],
                   **{'length_includes_head': True,
                      'width': 0.005 ,
                      'head_width':0.02,
                      'fc': 'b',
                      'ec': 'none'})

    fig.suptitle(figtitle, fontsize=15)
    ax.set_title('\nMakespan: ' + format(makespan(drone_info, used_drones, package_trail),'.5f

    startx, endx = ax.get_xlim()
    starty, endy = ax.get_ylim()


    ax.tick_params(which='both', # Options for both major and minor ticks
                top='off', # turn off top ticks
                left='off', # turn off left ticks
                right='off',  # turn off right ticks
                bottom='off') # turn off bottom ticks

    # Customize the major grid
    ax.grid(which='major', linestyle='-', linewidth='0.1', color='red')
    ax.grid(which='minor', linestyle=':', linewidth='0.1', color='black')

    #ax.xaxis.set_ticks(np.arange(startx, endx, 0.4))
    #ax.xaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f'))

    #ax.yaxis.set_ticks(np.arange(starty, endy, 0.4))
    #ax.yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f'))

    #plt.yticks(fontsize=5, rotation=90)
    #plt.xticks(fontsize=5)

    # A light grid
    #plt.grid(color='0.5', linestyle='--', linewidth=0.5)
```

◇

Fragment referenced in 3.

Todo list