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

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.
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]

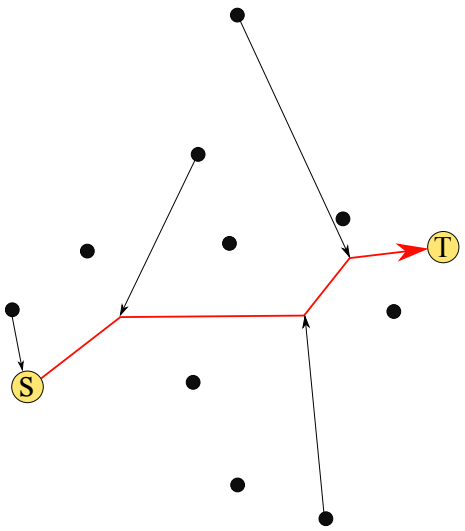


Figure 1.1: An instance of the Package Handoff problem for a single package being transported from S to T. Agents are located at the dots marked in black. The package travels along the red path. The agents all have different velocities, and in this example, assumed to have infinite battery capacity.

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.

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

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 [6] 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 6, ... ⟩
⟨ Run Handlers 29 ⟩
⟨ Plotting 36 ⟩
◇
```

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 Problem: 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.

[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

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 $H_{i_1} \dots H_{i_k}$ for $0 \leq i_0, \dots i_k \leq n$ stand for points where the drones with labels $i_0, \dots i_k$ 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.

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.

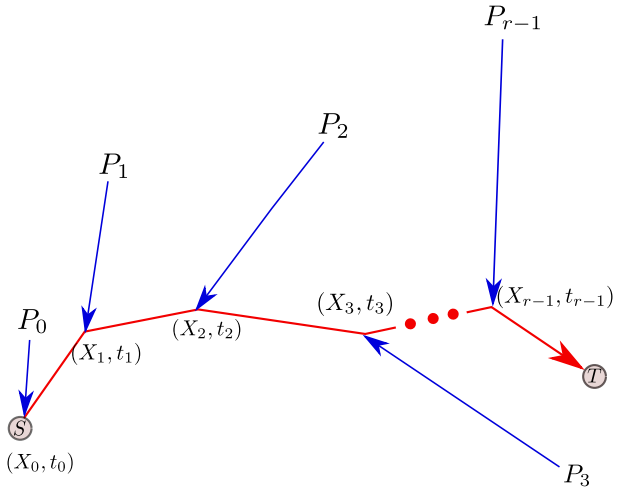


Figure 1.2: The path of the package is shown in red. The drones invovled in the handoff are labelled P_i in the prespecified handoff order.

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

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 6* ⟩ ≡

```
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 defined by 6, 7, 10a, 12a, 14a, 16.
Fragment referenced in 3.

We next describe a heuristic that use Continuous Dijkstra [11] type approach in computing approximate solutions to OPT.

1.3.2 Algorithm: 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.

Here is a sketch of the implementation of `algo_odw`. Use Figure 1.3.2 as a reference while reading the description below.

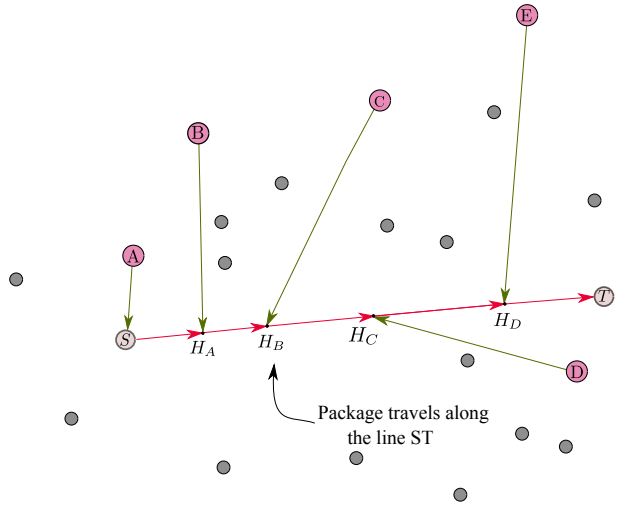


Figure 1.3: first figure

Figure 1.4: 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.

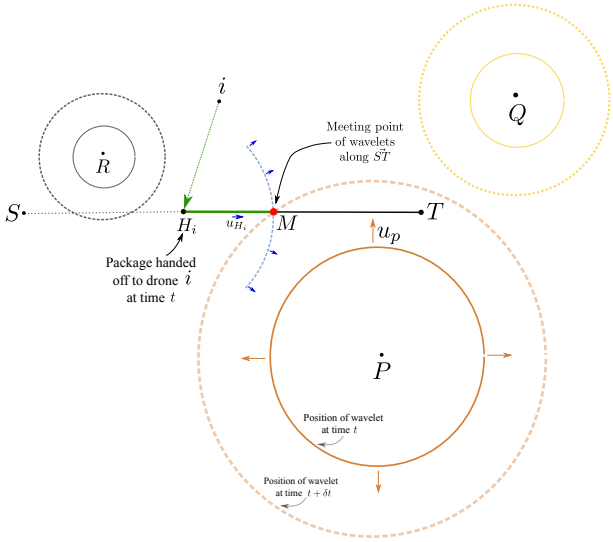


Figure 1.5: second figure

Figure 1.6: Intersection of two expanding wavelets along $H_i\vec{T}$. The figure shows an overlay of 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 it at at M)

⟨ *Algorithms* 7 ⟩ ≡

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

    from scipy.optimize import minimize
    source     = np.asarray(source)
    target     = np.asarray(target)
    sthat      = (target-source)/np.linalg.norm(target-source)
    numdrones  = len(drone_info)
    clock_time = 0.0  # time on the global clock

    ⟨ Find the drone which can get to the source the quickest 8a ⟩
    package_reached_p      = False

    while not(package_reached_p):
        ⟨ Find a faster wavelet that meets up with the package wavelet along line ST at the earliest 8b ⟩
        ⟨ Check if package wavelet reaches target before meeting wavelet computed above. Update states accordingly 11a ⟩

    ⟨ Run the convex optimization solver to retrieve the exact tour package_trail_cvx for given drone order 11c ⟩
    ⟨ Plot tour if plot_tour_p == True 12b ⟩

    return used_drones, package_trail_straight, mspan_straight, package_trail_cvx, mspan_cvx
```

◇

Fragment defined by 6, 7, 10a, 12a, 14a, 16.
Fragment referenced in 3.
Defines: `algo_odw` 29.

At the start of the algorithm, the package is stationary. We find the drone which can reach it at the earliest, and update the time on the global clock. It is easy to see that this drone must be involved in the optimal handoff schedule.

⟨ *Find the drone which can get to the source the quickest* 8a ⟩ ≡

```
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

clock_time = tmin

current_package_handler_idx = imin
current_package_position    = source

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

◇

Fragment referenced in 7.
Uses: `time_of_travel` 12a.

In the optimal handoff order, as we have already noted, the handoff happens from a slower to a faster robot. In the next chunk, we calculate the wavelet that meets the package wavelet at the earliest, subject to the constraint that the wavelet should be faster than the package wavelet.

⟨ *Find a faster wavelet that meets up with the package wavelet along line* $\vec{ST}$ *at the earliest* 8b ⟩ ≡

```
tI_min     = np.inf
idx_tI_min = None
for idx in drone_pool:

      us = drone_info[current_package_handler_idx][1]
      up = drone_info[idx][1]

      if up <= us: # slower drones are useless, so skip rest of the iteration
          continue
      else:
        s = current_package_position
        p = np.asarray(drone_info[idx][0])

        tI, x = get_interception_time_and_x(s, us, p, up, target, clock_time)

        if tI < tI_min:
           tI_min     = tI
           idx_tI_min = idx
```

◇

Fragment referenced in 7.
Uses: `get_interception_time_and_x` 10a.

We now implement `get_interception_time_and_x`, the time on the global clock when two wavelets meet and position along the line $\vec{ST}$. More precisely, the function computes the time on the global clock at which a wavelet that started expanding from $P = (\alpha, \beta)$ at time 0, meets a wavelet that started expanding from S at time t_0 along the half-line $\vec{ST}$. We

make a change of coordinages such that $\vec{ST}$ is horizontal and pointing to the right, and $S = (0, 0)$ as in Figure 1.3.2.

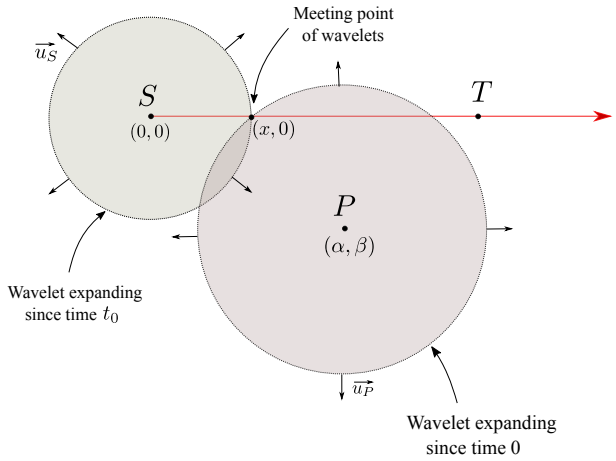


Figure 1.7: Reference figure for the function `get_interception_time_and_x`*. Without loss of generality (by changing the coordinate system) we can assume the line $\vec{ST}$ to be horizontal.*

$$\frac{x-0}{u_S} = \frac{\sqrt{(x-\alpha)^2+\beta^2}}{u_P} - t_0$$
$$\left(\frac{x}{u_S} + t_0\right)^2 = \frac{(x-\alpha)^2+\beta^2}{u_P^2}$$

Rearranging the terms, we get a quadratic equation in x that we can solve solve using standard non-linear solvers [5].

$$x^2\left(\frac{1}{u_S^2} - \frac{1}{u_P^2}\right) + x\left(\frac{2t_0}{u_S} + \frac{2\alpha}{u_P^2}\right) + \left(t_0^2 - \frac{\alpha^2}{u_P^2} - \frac{\beta^2}{u_P^2}\right) = 0 \tag{1.1}$$

Once we obtain x getting the interception time t_I (i.e. the time on the global clock when the two wavelets meet) as $t_I = \frac{x}{u_S} + t_0$.

[5] Using the exact formula for solving the quadratic equation is unstable in computer arithmetic as is well known from numerical analysis. To avoid such tricky issues, it is best to use the standard `roots` solver for polynomials available in NumPy https://docs.scipy.org/doc/numpy/reference/generated/numpy.roots.html

⟨ *Algorithms* 10a ⟩ ≡

```
def get_interception_time_and_x(s, us, p, up, t, t0) :

    ⟨ Change coordinates to make s = (0,0) and t to lie along X-axis as in ?? 10b ⟩

    # Solve quadratic equation as documented in main text
    qroots = np.roots([ (1.0/us**2 - 1.0/up**2),
                        2*t0/us + 2*alpha/up**2 ,
                        t0**2 - alpha**2/up**2 - beta**2/up**2])

    # The quadratic should always have a root.
    qroots = np.real(qroots) # in case the imaginary parts are really small
    qroots.sort()

    x = None
    for root in qroots:
        if root > 0.0:
           x = root
           break

    assert abs(x/us+t0 - np.sqrt((x-alpha)**2 + beta**2)/up) <= 1e-6 , \
           "Quadratic not solved perfectly"

    tI = x/us + t0
    return tI, x
```

◇

Fragment defined by 6, 7, 10a, 12a, 14a, 16.
Fragment referenced in 3.
Defines: `get_interception_time_and_x` 8b.

Massage the input into such that $\vec{ST}$ lies along the positive X-axis as in **??**.

⟨ *Change coordinates to make* $s = (0,0)$ *and* t *to lie along X-axis as in* **??** 10b ⟩ ≡

```
t_m = t - s # the _m subscript stands for modify
t_m = t_m / np.linalg.norm(t_m) # normalize to unit

# For rotating a vector clockwise by theta,
# to get the vector t_m into alignment with (1,0)
costh = t_m[0]/np.sqrt(t_m[0]**2 + t_m[1]**2)
sinth = t_m[1]/np.sqrt(t_m[0]**2 + t_m[1]**2)

rotmat = np.asarray([[costh, sinth],
                     [-sinth, costh]])

assert np.linalg.norm((rotmat.dot(t_m) - np.asarray([1,0]))) <= 1e-6,\
       "Rotation matrix did not work properly. t_m should get rotated\
        onto [1,0] after this transformation"

p_shift  = p - s
p_rot    = rotmat.dot(p_shift)
[alpha, beta] = p_rot
```

◇

Fragment referenced in 10a.

If the package reaches the target before it meets any other wavelet along the line $\vec{ST}$, then there is no point in handing

off the package to some other drone. Just terminate the handoff! Otherwise handoff the package and update variables accordingly.

⟨ *Check if package wavelet reaches target before meeting wavelet computed above. Update states accordingly* 11a ⟩ ≡

```
time_to_target_without_handoff = np.linalg.norm((target-current_package_position))/ \
                                 drone_info[current_package_handler_idx][1]

if time_to_target_without_handoff < tI_min :
     package_reached_p = True
     package_trail_straight.append(target)

else:
     ⟨ Update package information (current speed, position etc.) and drone information (available and used drones) 11b ⟩
```

◇

Fragment referenced in 7.

⟨ *Update package information (current speed, position etc.) and drone information (available and used drones)* 11b ⟩ ≡

```
package_handler_speed    = drone_info[current_package_handler_idx][1]
current_package_position = current_package_position + \
                           package_handler_speed * (tI_min - clock_time) *  sthat
package_trail_straight.append(current_package_position)

clock_time                  = tI_min
current_package_handler_idx = idx_tI_min

drone_pool.remove(idx_tI_min)
used_drones.append(idx_tI_min)
```

◇

Fragment referenced in 11a.

Now that we have a list of drones involved in the handoff, (along with an approximate trail for the package) we use the convex optimization solver to extract the exact tour for the given set of drones. We plot both tours for a visual comparison, if `plot_tour_p` is set to `True`.

⟨ *Run the convex optimization solver to retrieve the exact tour* `package_trail_cvx` *for given drone order* 11c ⟩ ≡

```
package_trail_cvx =  algo_pho_exact_given_order_of_drones(\
                           [drone_info[idx] for idx in used_drones],source,target)

mspan_straight    = makespan(drone_info, used_drones, package_trail_straight)
mspan_cvx         = makespan(drone_info, used_drones, package_trail_cvx)
```

◇

Fragment referenced in 7.

The next chunk implements a function that computes the makespan of the delivery process, i.e. the time it takes for the package to get from the source to the destination, given the points on the trajectory of the package and the drones involved alongwith the handoff order. The variable `drone_info` is list of tuples, where the $\mathtt{i}^{\text{th}}$ tuple gives the initial position and speed of the drone in the zeroth and first position respectively. The function `time_of_travel` is simply a function used to compute the time it takes for an agent with uniform speed to travel between a given `source` and `target`. Both `source` and `target` are numpy arrays of size two (if not, they are converted into numpy arrays at the start of the function).

⟨ *Algorithms* 12a ⟩ ≡

```
def time_of_travel(start, stop, speed):
     start = np.asarray(start)
     stop  = np.asarray(stop)
     return np.linalg.norm(stop-start)/speed

def extract_coordinates(points):

    xs, ys = [], []
    for pt in points:
        xs.append(pt[0])
        ys.append(pt[1])
    return np.asarray(xs), np.asarray(ys)

def makespan(drone_info, used_drones, package_trail):

    assert len(package_trail) == len(used_drones)+1, ""
    makespan = 0.0
    counter  = 0
    for idx in used_drones:
         dronespeed    = drone_info[idx][1]

         makespan += time_of_travel(package_trail[counter],\
                                    package_trail[counter+1],
                                    dronespeed)
         counter += 1

    return makespan
```

◇

Fragment defined by 6, 7, 10a, 12a, 14a, 16.
Fragment referenced in 3.
Defines: `extract_coordinates` 36, `makespan`, Never used, `time_of_travel` 8a.

⟨ *Plot tour if* `plot_tour_p == True` 12b ⟩ ≡

```
if plot_tour_p:
     fig0, ax0 = plt.subplots()
     plot_tour(fig0, ax0, "ODW: Straight Line", source, target, \
               drone_info, used_drones, package_trail_straight)

     fig1, ax1 = plt.subplots()
     plot_tour(fig1, ax1, "ODW: Straight Line, Post Convex Optimization", source, target, \
               drone_info, used_drones, package_trail_cvx)
     plt.show()
```

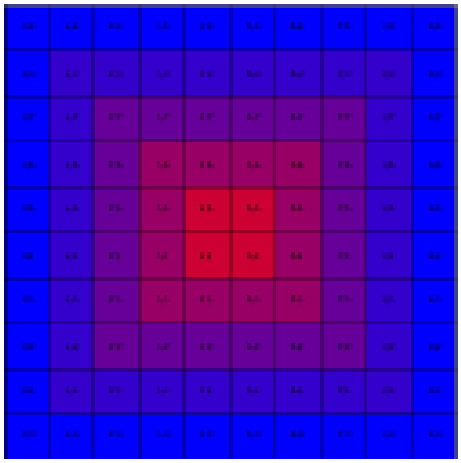
◇

Fragment referenced in 7.

Chapter 2

Multiple Package Handoff

What's more fun than delivering a single package? Delivering multiple packages! The moment, we generalize from a single to multiple packages the problem becomes enormously more interesting along with several possible generlizations to the statement of the problem. Each section is dedicated to one such generalization along with a description of algorithms and heuristics to solve them.



2.1 Problem: Drone Assignment To Packages

We are given as input the initial locations P_i of n agents in $\mathbb{R}^2$ each capable of a maximum speed $u_i > 0$. Also given are $k \leq n$ source target pairs S_j, T_j where S_j for $j \leq k$ denotes the beginning position of a package that has to get to target point T_j. Each agent is allowed to be involved in the transport of at most one package. More than one agent is allowed to be assigned to a package. Agents assigned to a package can coordinate to rendezvous and relay the package from its source to its destination (exactly as in the single package handoff case)

The objective is to perform an assignment of agents to the packages so that the time taken to deliver the last package to its target is minimized, i.e. the makespan of the deliveries of the packages from sources to their destinations is minimized. See for an example instance (alongside tours computed for the packages and drones by the "match and move" algorithm to be described next.)

2.1.1 Algorithm: Match-and-Move

The approach taken here is again based on Continuous Dijkstra. At the start of the algorithm, we assume each of the packages are constrained to travel along the the straight line segment joining its source and target. We then imagine wavelets expanding at speed u_i from each of the drone positions and perform an incremental assignment of the drones to the packages as the wavelet expansion proceeds. The assignment is done via bottleneck matching in an appropriately constructed bipartite graph. The matching algorithm is run everytime a certain "event" is detected.

Once the final assignment of drones to packages has been performed, we run the convex optimization solver desribed in subsection 1.3.1 to get the exact trajectory for each of the packages for the given assignment.

The rest of this section is devoted to making the above description more precise. First we give an outline of `algo_matchmove` that will be fleshed out in subsequent subsections.

⟨ *Algorithms* 14a ⟩ ≡

```
import networkx as nx

def algo_matchmove(drone_info, sources, targets, plot_tour_p = False):

    ⟨ Sanity checks on input for algo_matchmove 14b ⟩
    ⟨ Basic setup 17 ⟩

    while not all(package_delivered_p):
        ⟨ Construct bipartite graph G on drone wavelets and package wavelets 19 ⟩
        ⟨ Get a bottleneck matching on G 20a ⟩
        ⟨ Expand drone wavelets till an event of either Type I or Type II is detected 20b ⟩
        ⟨ Update drone pool and package states 20c ⟩

    ⟨ Run convex optimization solver to get improved tours for drones and packages 21a ⟩
    ⟨ Plot movement of packages and drones if plot_tour_p == True  21b ⟩
    #return pass pass pass pass pass
```

◇

Fragment defined by 6, 7, 10a, 12a, 14a, 16.
Fragment referenced in 3.
Defines: `algo_matchmove` 29.

2.1.1.1 Sanity Checks, and Basic Setup

To start things off, we enforce the constraint that the number of drones should be greater than the number of packages. Also the number of packages is equal to the number of sources which in turn is equal to the number of targets. These conditions are encoded as sanity checks in the code chunk below that is incorporated at the beginning of the function

⟨ *Sanity checks on input for* `algo_matchmove` 14b ⟩ ≡

```
assert len(drone_info) >= len(sources),\
   "Num drones should be >= the num source-target pairs"

assert len(sources) == len(targets),\
   "Num sources should be == Num targets"
```

◇

Fragment referenced in 14a.

The lists `sources`, `targets`, `drone_initposns` and `drone_speeds` are all *constant* throughout the execution of the algorithm. They denote the list of sources, targets, initial position and speeds of the drones respectively. For package `i`, its source and destination are respectively `source[i]` and `target[i]`.

Then, we create some lists that keep track of various states in the main `while` loop of the algorithm.

A. **`package_delivered_p`** is a Boolean list where the `i`$^{\text{th}}$ element keeps track whether the package with id `i` has been delivered to its destination. The main `while` loop stops when all flags are set to `True`.

B. **`drone_pool`** is a list of ids of drones that can be considered for the bottleneck matching process for the next iteration of the `while` loop.

C. **`drone_wavelet_info`** is a list of list of dictionaries. Each outer list corresponds to a list of wavelets corresponding to each drone. Every wavelet is represented by a dictionary, with three attributes

1. **`wavelet_center`** The cartesian coordinates of the center H of the wavelet about which the wavelet expands.
2. **`clock_time`** The time on the global clock at which the wavelet started expanding around the wavelet center H.
3. **`matched_package_ids`** A list of the ids of the packages that were assigned to the wavelet during its expansion around center H. Each such list is initialized as the empty list.

See Figure 2.1.1.1 to see a sequence of wavelets corresponding to a single drone.

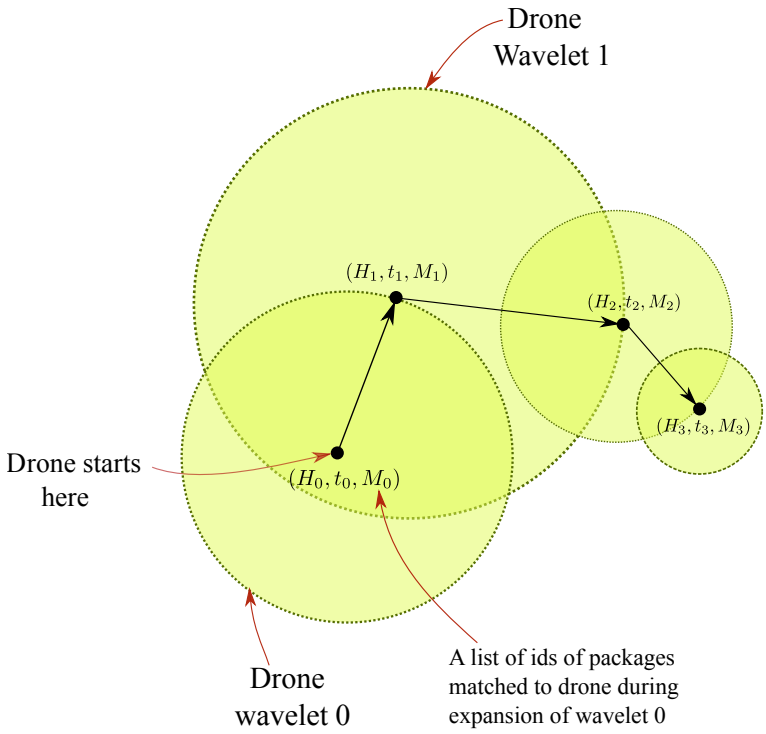


Figure 2.1: A sequence of wavelets corresponding to a drone which is stored as a list of dictionaries (each dictionary corresponding to a wavelet) `[{'wavelet_center':`H_i`, 'clock_time':`t_i`, 'matched_package_ids':` M_i`}]`*. Each wavelet expands around H_i at speed u_i starting at time t_i on the global clock. Every drone has a sequence of wavelets. The H_i also represent handoff points where the drone either hands a package off to another drone or is handed a package by another drone. Note that the usage of the letter H to represent the center of a drone wavelet is suggestive of it also being a point where a package is handed to the drone or a package is handed off to another drone.*

D. **`package_trail_info`** is a list of list of dictionaries, similar to `drone_wavelet_info`. The `i`$^{\text{th}}$ element of the outer list corresponds to the trail of a package. In each trail (itself a list), we record the positions of a package as well as the time on the clock t at which that position was recorded. We also note the id of the last drone known to have handled the package at time $t - \varepsilon$ for ε being an arbitrarily small positive number.

Two helper functions `get_current_position_of_package` and `get_current_speed_of_package` to extract information conveniently from `package_trail_info`. Again like `get_last_wavelet_of_drone` the result returned depends on the state of `package_trail_info` at the time the function is called.

We will also need a slightly generalized version of `get_interception_time_and_x` that takes into account when a wavelet started expanding in order to calculate x and the time t_I on the global clock when the wavelets will meet.

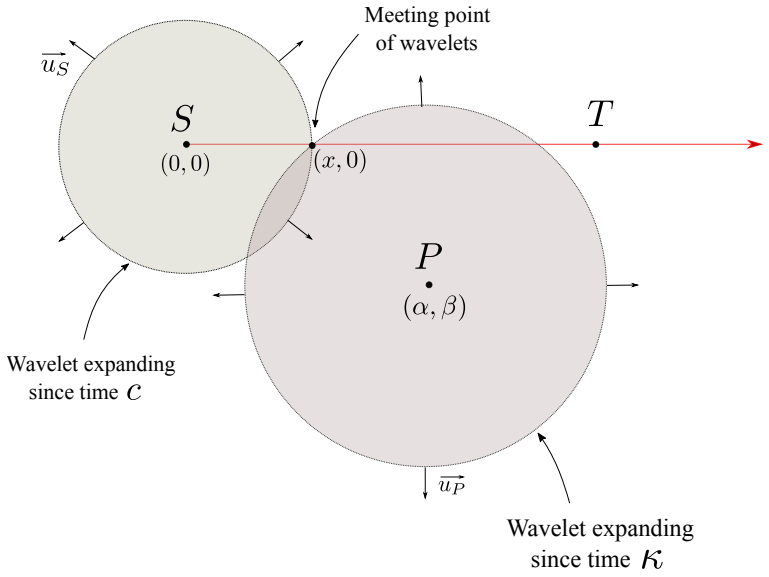


Figure 2.2: Without loss of generality (by changing the coordinate system) we can assume the line $\vec{ST}$ to be horizontal. We would like to compute the time t_I on the global clock when the two wavelets meet. The package wavelet starts expanding at time c which is the same as the global clock time at the moment the function is called and the dronw wavelet at time κ.

⟨ *Algorithms* 16 ⟩ ≡

```
def get_interception_time_and_x_generalized(s, us, p, up, t, c, k) :

    # Change coordinates
    t_m = t - s # the _m subscript stands for modify
    t_m = t_m / np.linalg.norm(t_m) # normalize to unit

    # For rotating a vector clockwise by theta,
    # to get the vector t_m into alignment with (1,0)
    costh = t_m[0]/np.sqrt(t_m[0]**2 + t_m[1]**2)
    sinth = t_m[1]/np.sqrt(t_m[0]**2 + t_m[1]**2)

    rotmat = np.asarray([[costh, sinth],
                         [-sinth, costh]])

    assert np.linalg.norm((rotmat.dot(t_m) - np.asarray([1,0]))) <= 1e-6,\
       "Rotation matrix did not work properly. t_m should get rotated\
        onto [1,0] after this transformation"

    p_shift  = p - s
    p_rot    = rotmat.dot(p_shift)
    [alpha, beta] = p_rot

    # Solve quadratic equation as documented in main text
    qroots = np.roots([ , , ])

    assert 1==2, "Unimplemented error"
    return tI, x
```

◇

Fragment defined by 6, 7, 10a, 12a, 14a, 16.
Fragment referenced in 3.

⟨ *Basic setup* 17 ⟩ ≡

```
sources             = [np.asarray(source) for source in sources]
targets             = [np.asarray(target) for target in targets]

drone_initposns     = [ np.asarray(initposn) for (initposn, _) in drone_info ]
drone_speeds        = [ speed                for (_,    speed) in drone_info ]

numpackages         = len(sources)
numdrones           = len(drone_info)

package_delivered_p = [ False for _ in range(numpackages) ]
drone_locked_p      = [ False for _ in range(numdrones)   ]

drone_pool          = range(numdrones)
remaining_packages  = range(numpackages)
global_clock_time   = 0.0
#.....................................................................
drone_wavelets_info = [ [{'wavelet_center'        : posn,
                          'clock_time'            : 0.0,
                          'matched_package_ids'   : []}]
                        for posn in drone_initposns ]

def get_last_wavelet_of_drone(i):
        return drone_wavelets_info[i][-1]
#.....................................................................
package_trail_info  = [ [{'current_position'   : source,
                          'clock_time'         : 0.0,
                          'current_handler_id' : None }]
                        for source in sources ]

def get_current_position_of_package(i):
        return package_trail_info[i][-1]['current_position']

def get_current_speed_of_package(i):
        current_handler_id = package_trail_info[i][-1]['current_handler_id']

        if current_handler_id is None:
             return 0.0
        else:
             return drone_speeds[current_handler_id]

def get_current_handler_of_package(i):
        return package_trail_info[i][-1]['current_handler_id']

def dronelabel(idx):
    return 'drone_' + str(idx)

def packagelabel(idx):
    return 'package_' + str(idx)
```

◇

Fragment referenced in 14a.
Defines: `dronelabel`, Never used, `get_current_position_of_package`, Never used, `get_current_speed_of_package`, Never used, `get_last_wavelet_of_drone`, Never used, `packagelabel` 19.

The `get_wavelet_of_drone` function defined at the end of the above chunk is a convenience function for returning the

center of the last wavelet and the associated clock time recorded at that center for an arbitrary drone. **Note:** The result of calling this function depends on the time at which it is called because `drone_wavelet_info` list is mutated during the main while loop.

2.1.1.2 Main Loop

It is critical that the zero testing for upkg has been done for `time_target_to_solo` to be computed safely without worrying about `ZeroDivisionError`.

Note that the graph bipartite G is encoded as a matrix of weights as shown in the figure Figure 2.1.1.2

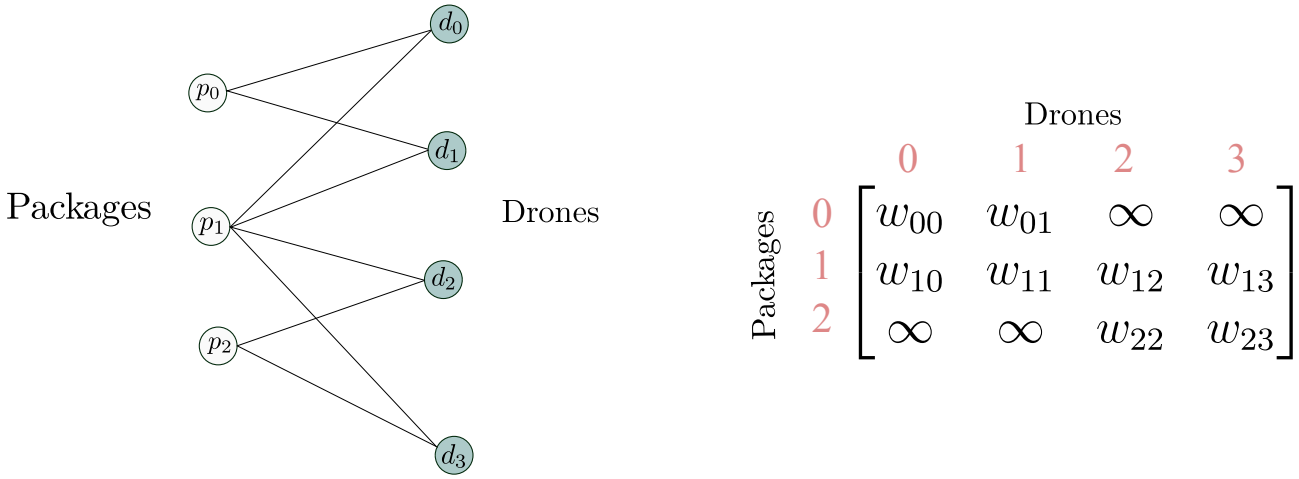


Figure 2.3: A bipartite graph G constructed between the drones and packages, along witht the asosciated weight matrix. Edges between drones and packages not in the graph are represented by infinite edge weights. A minimum weight bipartite matching algorithm from SciPy (`http://bit.ly/2MAWHPn`) will be used as a replacement for bottleneck matching (SciPy / NetworkX do not have such a solver) The matrix shown in the right half of the above figure is built inside the code, one column at a time.

⟨ *Construct bipartite graph G on drone wavelets and package wavelets* 19 ⟩ ≡

```
infty       = np.inf
G_mat       = np.full((len(remaining_packages),len(drone_pool)), infty)
lbend_edges = []

for didx in drone_pool:
    dlabel = dronelabel(didx)

    for pidx in remaining_packages:
        current_handler_of_package = get_current_handler_of_package(pidx)

        if current_handler_of_package != didx and not(drone_locked_p[didx]):

                plabel = packagelabel(pidx)
                target = targets[pidx]

                pkg  = get_current_position_of_package(pidx)
                upkg = get_current_speed_of_package(pidx)

                wav  = get_last_wavelet_of_drone(didx)
                dro  = wav['wavelet_center']
                udro = drone_speeds[didx]
                zerotol = 1e-7

                if upkg < zerotol :
                      G_mat[pidx, didx] = np.linalg.norm(pkg-dro)/udro +\
                                          np.linalg.norm(target-pkg)/udro
                      lbend_edges.append({'edge_pair': (pidx,didx),
                                          'y'        : np.linalg.norm(pkg-dro)/udro })

                elif udro > upkg and abs(upkg-udro) > zerotol:

                     time_to_target_solo = np.linalg.norm(target-pkg)/upkg
                     tI, x  = get_interception_time_and_x_generalized(pkg, upkg, dro, udro, target,
                                                                      global_clock_time, wav['clock_time'])
                     if x is not None:
                        assert tI is not None, "tI and x should be None or not None simultaneously"
                        if tI < global_clock_time + time_to_target_solo:

                          pthat             = (target-pkg)/np.linalg.norm(target-pkg)
                          interception_pt   = pkg + x * pthat
                          G_mat[pidx, didx] = tI + np.linalg.norm((target-interception_pt))/udro
                          lbend_edges.append({'edge_pair': (pidx,didx),
                                              'y'       : np.linalg.norm(interception_pt-dro)/udro })

        elif current_handler_of_package == didx and drone_locked_p[didx]:

               assert abs(udro-upkg) < zerotol , "udro should be equal to upkg"
               G_mat[pidx, didx] = np.linalg.norm((target-pkg))/udro

        elif current_handler_of_package != didx and drone_locked_p[didx]:
               pass # drone locked, so it cant help
        else :
               # The outer not negates the inner condition which
               # is true if this branch is executed
               assert not(current_handler_of_package == didx and \
                          not(drone_locked_p[didx])) ,\
                      "This else branch should not be executed. This\
                       means didx is handling a package and is NOT locked"
```

◇

Fragment referenced in 14a.
Uses: packagelabel 17.

Since NetworkX does not have a native bottleneck matching solver, we use a minimum-weight matching solver already present in SciPy documented at http://bit.ly/2MAWHPn.

⟨ *Get a bottleneck matching on G* 20a ⟩ ≡

```
from scipy.optimize import linear_sum_assignment
pkg_ind, dro_ind = linear_sum_assignment(G_mat)

print "Lbend edges"
print lbend_edges
return
```

◇

Fragment referenced in 14a.

There are exactly two types of (mutually exclusive) events in this algorithm.

Type I: A package wavelet reaches its target.

Type II: A wavelet corresponding to a drone (not handling a package) meets up with a package wavelet.

To detect these, we keep track of a so-called list of "L bend" edges as shown in the figure below (see the variable `lbend_edges` in the previous chunk that keeps track of this quantity).

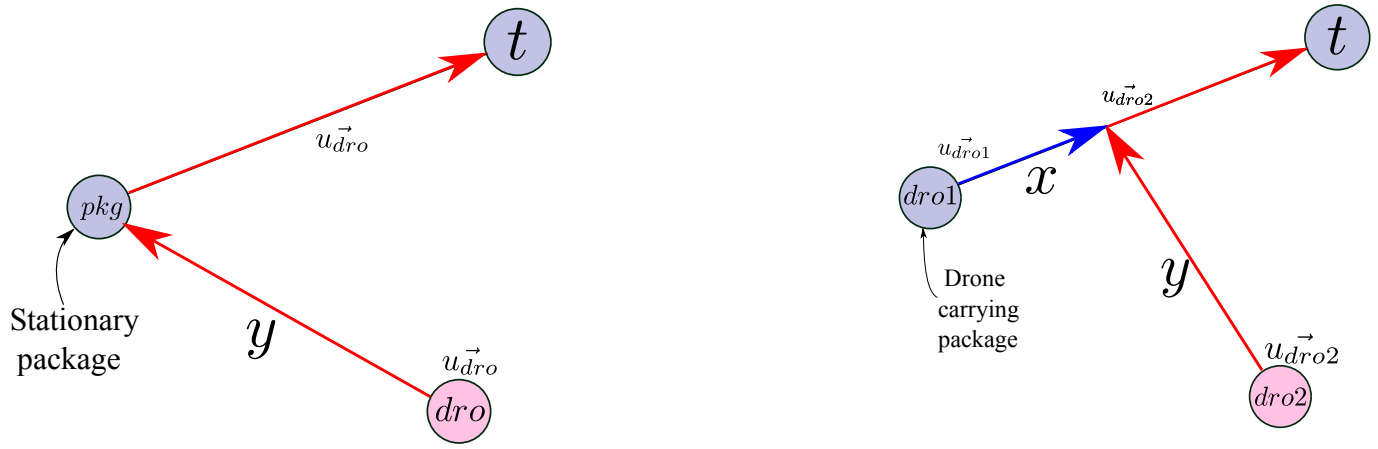


Figure 2.4: If a package drone pair is represented in the graph matrix as a finite weight edge, we first check if it is an L-bend edge that corresponds to one of the two situations above. A drone catches up with another drone carrying the package (right) or a drone picks up a stationary package and moves towards the target (left)

⟨ *Expand drone wavelets till an event of either Type I or Type II is detected* 20b ⟩ ≡

```
pass
```

◇

Fragment referenced in 14a.

⟨ *Update drone pool and package states* 20c ⟩ ≡

```
pass
```

◇

Fragment referenced in 14a.

⟨ *Run convex optimization solver to get improved tours for drones and packages* 21a ⟩ ≡

```
pass
```

◇

Fragment referenced in 14a.

⟨ *Plot movement of packages and drones if* `plot_tour_p == True` 21b ⟩ ≡

```
pass
```

◇

Fragment referenced in 14a.

Bibliography

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

[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] 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.

[5] *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).

[6] 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.

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

[9] 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.

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

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

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

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

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

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

[16] *"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).

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 [16] and [14] respectively.

A system of using relays for delivering packages is not a particularly new idea. A famous (and shortlived!) 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[15] [5] and Matternet [10] 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 [9]

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 [12]

> *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 tiffins return to suburban homes by 6.00 pm.*

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

[1] literally: lunchbox carriers

Appendix B

README.md

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

"README.md" 25≡

```
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" 26≡

```
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" 27≡

```
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 a

        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)
◇
```

Appendix E

Implementation of ⟨Run Handlers⟩

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

⟨ *Run Handlers* 29 ⟩ ≡

```
# 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)

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, plot_tour_p=False):
           return algo( self.drone_info,
                        self.source,
                        self.target,
                        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 the drone at '+str(ne
                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', lw=1.0 ))
                 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', lw=1.0 ))
                 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 clear them
               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'

    xlim = utils_graphics.xlim
    ylim = utils_graphics.ylim

    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()




class Multiple_PHO_Input:
    def __init__(self, drone_info = [] , sources = [], targets=[]):
           self.drone_info = drone_info
           self.sources     = sources
           self.targets     = targets

    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, plot_tour_p=False):
           return algo( self.drone_info,
                        self.sources,
                        self.targets,
                        plot_tour_p    )

    # Methods for \verb|ReverseHorseflyInput|
    def clearAllStates (self):
          self.drone_info = []
          self.sources = []
          self.targets = []
```

```
def multiple_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 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 the drone at '+str(ne
                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='#EBEBEB', edgecolor='black'  ))

                ax.text( newPoint[0], newPoint[1], "{:.2f}".format(speed), fontsize=10,
                         horizontalalignment='center', verticalalignment='center' )

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

            elif event.button == 3:
                # distinct colors, obtained from https://sashat.me/2017/01/11/list-of-20-simple-distinct-colors/
                cols = ['#e6194b', '#3cb44b', '#ffe119', '#4363d8', '#f58231',
                '#911eb4', '#46f0f0', '#f032e6', '#bcf60c', '#fabebe',
                '#008080', '#e6beff', '#9a6324', '#fffac8', '#800000',
                '#aaffc3', '#808000', '#ffd8b1', '#000075', '#808080']

                # Insert big colored circles representing the source and target points

                patchSize  = (xlim[1]-xlim[0])/50.0
                if (len(run.sources) + len(run.targets)) % 2 == 0 :
                       run.sources.append((event.xdata, event.ydata))
                       ax.add_patch( mpl.patches.Circle( run.sources[-1], radius = patchSize,
                                                         facecolor= cols[len(run.sources) % len(cols)], edgecolor='black'
                       ax.text( run.sources[-1][0], run.sources[-1][1], 'S'+str(len(run.sources)), fontsize=15,
                                horizontalalignment='center', verticalalignment='center' )

                else :
                     run.targets.append((event.xdata, event.ydata))
                     ax.add_patch( mpl.patches.Circle( run.targets[-1], radius = patchSize,
                                                       facecolor= cols[len(run.sources)%len(cols)], edgecolor='blac
                     ax.text( run.targets[-1][0], run.targets[-1][1], 'T'+str(len(run.targets)), fontsize=15,
                              horizontalalignment='center', verticalalignment='center' )

            # 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"   +\
                         " (mm)     Match-and-move \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 == 'mm':
                       tour = run.get_tour( algo_matchmove, 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 = Multiple_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'

xlim = utils_graphics.xlim
ylim = utils_graphics.ylim

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, sources and targets onto canvas.")

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.
Uses: algo_matchmove 14a, algo_odw 7.

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.

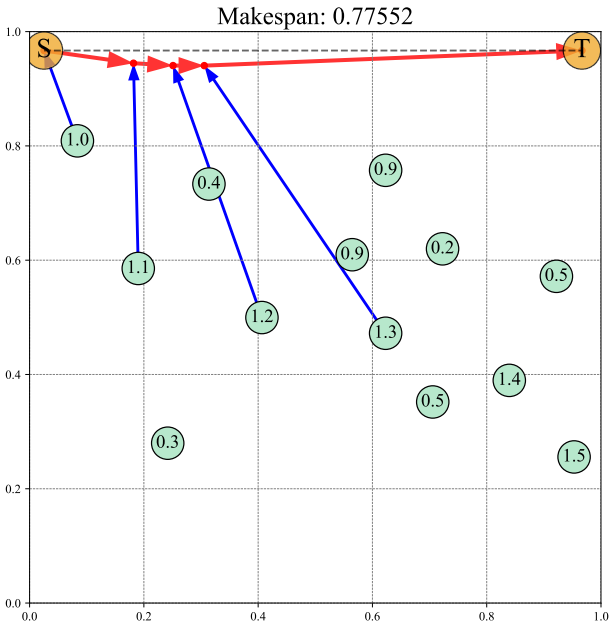


Figure F.1: The numbers inside the circle indicate the speed of the drone

⟨ *Plotting* 36 ⟩ ≡

```
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=10,
              speedmarkersize=20,
              sourcetargetmarkerfontsize=15,
              sourcetargetmarkersize=20 ):
```

```
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':0.025,
                   '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=1.0, ms=10, mec='k', mfc='#F1AB30' )
#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 = 1.0, 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],'.2f'),
              fontsize=speedfontsize, horizontalalignment='center', verticalalignment='center' )

# 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'), fontsize=16)

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.
Uses: extract_coordinates 12a.

Todo list