Horsefly Problems on Graphs

Saturday 1st June, 2019

Contents

Chapter 1

Introduction

This set of literate programs implement heuristics to solve Horsefly type problems on graphs. The generic problem for this setting is: given a set of nodes, with some distinguished nodes marked as delivery points and one node marked as the starting point of a truck and drone we want to minimize the makespan of the delivery process in which the drone departs from the truck with a package, flies to a site along a sequence of edges, drops of the package and returns to the truck (which itself might be moving along the graph edges) to pick up the next package for delivery. For the sake of simplicity, we only allow the drone itself to do the delivery and the meeting points between the truck and drone to happen at the nodes of the graph. Some edges in the graph will be traversable by only the truck or only the drone and some edges will be traversable by both.

The challenge in this problem is to set up a coordinated schedule of the truck and the drone so that the total delivery time is minimized. The truck is assumed to always travel with unit speed and the drone with integer speed $\varphi \in \{2,3,\ldots\}$.

While almost all the algorithms developed here are intended to be graph agnostic, the input graphs for such problems will typically come about from the discretization of geometric domains where there might be obstacles, some for the drone, some for the truck, some for maybe both. To test the algorithms interactively, I will typically use rectangular obstacles, along with an appropriate background mesh of points.

Several variants on this basic theme are possible in which a truck might have been equipped with more than one drone or there might be constraints at the sites or pick-up points (such as service or package-handoff times). Lots of special cases abound too: what if the underlying graph is or outerplanar planar, what if the set of truck and/or drone edges form a tree? What if the truck is restricted to travelling only along a path-graph?

Heuristics for such variants and their experimental-performance analyses will also be discussed later. Such problems are typically extremely hard, so all heuristics are meant to be approximations to the optimal routing scheme for each instance. Algorithms developed will typically use a combination of greedy strategies, dynamic programming, branch-and-bound and local search. Several interesting problems also arise in being able to efficiently implement a particular strategy and so answers to these questions will often be developed in tandem with the implementation of different strategies for Horsefly problems.

Chapter 2

User Input

This chapter implements (boring but important) UI code for the user to mouse in rectangular obstacles, service points, and initial position of the truck and drone onto the Matplotlib canvas and generate an appropriate geometric graph for the truck and drone to travel along. Typically these geometric graphs will be the visibility graph, delaunay triangulation or some other graph on such points which can be set at run-time through a command-line menu. The resulting algorithms, however, will typically be graph-agnostic unless explicitly indicated in the doc-string of the implemented algorithm.

One can also enter in the set of truck and drone obstacles along with any discretization parameters via a YAML file. This is particularly useful for storing nice instances and answers discovered wihle experimenting on a canvas.

Here is a high-level overview of the `horsefly_graphs.py` module.

"horsefly_graphs.py" 3≡

```
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import rc
import numpy as np
import scipy as sp
import sys, os, time
import itertools
from colorama import Fore, Style
import logging
import utils_graphics, utils_algo
import networkx as nx
from CGAL.CGAL_Kernel import Point_2, Segment_2, Iso_rectangle_2
from CGAL.CGAL_Kernel import do_intersect, intersection

⟨ Local data-structures 4, ... ⟩
⟨ Some basic canvas functions 6a⟩
⟨ Implementation of the key-press handler 7a⟩
⟨ Implementation of wrapperEnterPoints 7b⟩
⟨ Implementation of wrapperHoverObstacle 8⟩
⟨ Implementation of wrapperPlaceObstacle 9⟩
⟨ Implementation of wrapperResizeHoveringObstacle 10⟩

if __name__ == "__main__":
    ⟨ Body of main function 6b⟩
◇
```

Local Data Structures

The data-structure used by the canvas for manipulating the input is `HorseflyInputGraph`. It performs simple house-keeping duties such as keeping track of the sites, the initial position of the truck, various obstacles etc. The main method for this class is `getTour` which generates the actual graph according to the graph-policy specified and runs the appropriate algorithm on the generated graph. The flag `obstacle\_input\_mode\_p` is toggled to `True` or `False` according as obstacles are being entered onto the canvas. The input is entered in two modes (the default site insertion mode and the optional obstacle insertion mode, which is toggled by pressing the “o” or “O” key.)

The definition of the method `makeHorseflyInputGraph` will be given later. This will help exposition since details of the function depend on the `Obstacle` class which will be introduced in the next code-chunk.

⟨ *Local data-structures* 4 ⟩ ≡

```
class HorseflyInputGraph:
      def __init__(self, sites=[], inithorseposn=[]):
           self.sites               = sites
           self.inithorseposn       = inithorseposn
           self.obstacle_list       = []
           self.horsefly_input_graph = nx.Graph()

           # Tracking variables used only during interactive input.
           # To be frank these tracking variables should be placed in a separate class
           # they seem out of place here. Else have a dedicated class stored as a
           # variable/dictionary containinig all these icky state variables used
           # during input manipulation.
           self.default_obstacle_width   = 0.1
           self.default_obstacle_height  = 0.1

           self.hovering_obstacle_width  = 0.1 # This will be modified during the move
           self.hovering_obstacle_height = 0.1 # This will be modified during the move

           self.horse_obstacle_input_mode_p = False
           self.fly_obstacle_input_mode_p = False
           self.common_obstacle_input_mode_p = False


      def clearAllStates (self):
          self.sites           = []
          self.inithorseposn   = None
          self.obstacle_list = []

          self.horse_obstacle_input_mode_p  = False
          self.fly_obstacle_input_mode_p    = False
          self.common_obstacle_input_mode_p = False

          self.horsefly_input_graph = nx.Graph()

      ⟨ Definition of method makeHorseflyInputGraph 12 ⟩
      ⟨ Definition of method renderHorseflyInputGraph 13 ⟩

      def clearGraph(self,fig,ax):
          self.horsefly_input_graph = nx.Graph() # Input Graph is set to the empty null graph
          # TODO, clear the canvas completely and then redraw the current sites and obstacles stored
          # so that the drawn graph is cleared from the canvas, and the input is restored to its
          # original prsitine state, so that other algorithms can be tried out.
          pass

      def getTour(self, algo, phi):
             return algo(self.horsefly_input_graph, phi)
```

◇

Fragment defined by 4, 5.
Fragment referenced in 3.

The UI code allows only rectangular obstacles to be inserted for convenience of implementation. As has already been mentioned, once the underlying domain of obsctales and surrounding space is discretized the algorithms are graph agnostic. Code for inserting obstacles of more complicated shapes will be added later.

⟨ *Local data-structures* 5 ⟩ ≡

```
class Obstacle:
    def __init__(self, llcorner, width, height, figure=None,
                 axes_object=None, obstype=None, diskcolor = 'crimson'):
        self.llcorner = llcorner
        self.width    = width
        self.height   = height
        self.fig      = figure
        self.ax       = axes_object
        self.obstype  = obstype

        # Only inserted disks will end up having this attribute
        if self.fig != None:
             self.canvas_patch =  mpl.patches.Rectangle( self.llcorner, self.width,
                                                         self.height  , facecolor = diskcolor,
                                                         alpha = 0.7 )

    def mplPatch(self, diskcolor= 'crimson' ):
        return  self.canvas_patch

    def getVertices(self):
         [x,y] = self.llcorner
         p_ll = [x            , y       ]
         p_lr = [x+self.width , y       ]
         p_ur = [x+self.width , y+self.height]
         p_ul = [x            , y+self.height]
         return [p_ll,p_lr,p_ur,p_ul]

    def intersectionWithSegment(self,p,q):

        [llv, _, urv, _] = self.getVertices()
        llv = Point_2(llv[0],llv[1])
        urv = Point_2(urv[0],urv[1])
        rect = Iso_rectangle_2(llv,urv)

        p = Point_2(p[0],p[1])
        q = Point_2(q[0],q[1])
        seg = Segment_2(p,q)

        interscn_object = intersection(rect,seg)
        interscn_p     = do_intersect(rect,seg)

        return (interscn_p, interscn_object)
◇
```

Fragment defined by 4, 5.
Fragment referenced in 3.

Some basic canvas functions

A few basic functions are needed to manipulate the matplotlib canvas, such as clearing the canvas of patches, setting aspect ratio etc. We implement them here.

⟨ *Some basic canvas functions* 6a ⟩ ≡

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

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)

◇
```

Fragment referenced in 3.

The `main` function just sets up the basic canvas and the container to hold the user-input data. Once the canvas is started, it "listens in" on the mouse-press and key-press events to enter various elements such as the initial position of the truck and drone, the sites of delivery, the input obstacles.

⟨ *Body of main function* 6b ⟩ ≡

```
fig, ax =  plt.subplots()
run = HorseflyInputGraph()

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

mouseClick = wrapperEnterPoints (fig,ax, run)
fig.canvas.mpl_connect('button_press_event' , mouseClick )

keyPress   = wrapperkeyPressHandler(fig,ax, run)
fig.canvas.mpl_connect('key_press_event', keyPress )

hoverObstacle  = wrapperHoverObstacle(fig,ax, run)
fig.canvas.mpl_connect('motion_notify_event', hoverObstacle )

placeObstacle  = wrapperPlaceObstacle(fig,ax, run)
fig.canvas.mpl_connect('button_press_event' , placeObstacle )

resizeHoveringObstacle  = wrapperResizeHoveringObstacle(fig,ax, run)
fig.canvas.mpl_connect('key_press_event', resizeHoveringObstacle)

plt.show()
◇
```

Fragment referenced in 3.

This function handles the manages events corresponding to key-presses on the computer key-board. For instance, pressing `Ctrl+` → increases (resp. decreases) the X-dimension of the rectangle. Similarly for the Y-dimension and the keys `Ctrl+` ↑ and `Ctrl+` ↓. Behavior of other-keys is self-explantory as documented in the code-chunks below.

⟨ *Implementation of the key-press handler* 7a ⟩ ≡

```
def wrapperkeyPressHandler(fig,ax, run):
      def _keyPressHandler(event):

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

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

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

            elif event.key in ['h' , 'H']: # `h` for horse
                 run.horse_obstacle_input_mode_p    = not (run.horse_obstacle_input_mode_p)
                 run.fly_obstacle_input_mode_p      = False
                 run.common_obstacle_input_mode_p   = False


            elif event.key in ['f' , 'F']: # `f` for fly
                 run.horse_obstacle_input_mode_p   = False
                 run.fly_obstacle_input_mode_p     = not (run.fly_obstacle_input_mode_p)
                 run.common_obstacle_input_mode_p  = False


           elif event.key in ['g' , 'G']: # 'g' lies between the f and h keys on the keybard signifying intersection
                 run.horse_obstacle_input_mode_p   = False
                 run.fly_obstacle_input_mode_p     = False
                 run.common_obstacle_input_mode_p  = not (run.common_obstacle_input_mode_p)

            elif event.key in ['d','D']: # `d` for discretize domain, using the obstacles we sprinkle
                                         # some points and discretize everything
                 background_grid_pts = [[np.random.rand(), np.random.rand()] for i in range(60)]
                 run.makeHorseflyInputGraph(fig, ax, background_grid_pts)

      return _keyPressHandler
```

◇

Fragment referenced in 3.

This function just implements the behavior of the canvas under mouse-clicks. The left mouse button double-clicked inserts a site onto the canvas while the right mouse button inserts the intial position of the truck and drone. These two types of points have been given different colors. Later on the points used in the background grid to discretize the domain will again be shown in a different colour.

⟨ *Implementation of* `wrapperEnterPoints` 7b ⟩ ≡

```
def wrapperEnterPoints(fig,ax,run):
    def _enterPoints(event):
        if event.name      == 'button_press_event'          and \
           run.horse_obstacle_input_mode_p  == False        and \
           run.fly_obstacle_input_mode_p    == False        and \
           run.common_obstacle_input_mode_p == False        and \
```

```
            (event.button   == 1 or event.button == 3)       and \
             event.dblclick == True and event.xdata  != None and event.ydata  != None:

              if event.button == 1:
                  # Insert blue circle representing a site
                  newPoint = (event.xdata, event.ydata)
                  run.sites.append( newPoint  )
                  patchSize  = (utils_graphics.xlim[1]-utils_graphics.xlim[0])/140.0

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


              elif event.button == 3:
                  # Insert big red circle representing initial position of horse and fly
                  newinithorseposn  = (event.xdata, event.ydata)
                  run.inithorseposn = newinithorseposn
                  patchSize         = (utils_graphics.xlim[1]-utils_graphics.xlim[0])/100.0

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

                  print Fore.RED, "Initial positions of truck\n",
                  print run.inithorseposn
                  print Style.RESET_ALL

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

      return _enterPoints
◇
```

Fragment referenced in 3.

This function allows allows me to move a potential obstacle over the canvas before double-clicking it into place. It is called a hovering obstacle because the position of the rectangular obstacle follows the position of the mouse cursor as it moves over the matplotlib canvas. There are three obstacle modes one for truck-only obstacles (indicated by crimson) one for drone-only obstacles (indicated by blue) and one for obstacles common to both the truck and drone (indicated by green). To exit any of these obstacle placement modes to insert more sites, just press the corresponding obstacle mode keys again ('h', 'f' and 'g') to toggle them.

⟨ *Implementation of* `wrapperHoverObstacle` 8 ⟩ ≡

```
def wrapperHoverObstacle(fig,ax, run):
        """ Wrapper for the call-back function _hoverObstacle
        """
        def _hoverObstacle(event, previous_patch = []):
          """ Bind the motion of the mouse with the movement of a disk to be placed.
          """
          if previous_patch != []:
              previous_patch.pop().remove() # This physically removes the patch from the screen and from memory

          if event.xdata != None and event.ydata!=None and (run.horse_obstacle_input_mode_p  or
                                                            run.fly_obstacle_input_mode_p    or
                                                            run.common_obstacle_input_mode_p):
              if run.horse_obstacle_input_mode_p:
                    fcol = 'crimson'
```

```
                elif run.fly_obstacle_input_mode_p:
                       fcol = 'blue'

                elif run.common_obstacle_input_mode_p:
                       fcol = 'green'

                current_patch = mpl.patches.Rectangle((event.xdata,event.ydata),             \
                                                width     = run.hovering_obstacle_width,  \
                                                height    = run.hovering_obstacle_height,  \
                                                facecolor = fcol,   \
                                                alpha     = 0.5)
                previous_patch.append(current_patch)
                ax.add_patch(current_patch)

                fig.canvas.draw()
          return _hoverObstacle
    ◇
```

Fragment referenced in 3.

When you decide to place an obstacle at a particular place on the canvas, just double-click it into place.

⟨ *Implementation of* wrapperPlaceObstacle 9 ⟩ ≡

```
    def wrapperPlaceObstacle(fig,ax,run):
          def _placeObstacle(event):
              """ Double-clicking Button 1 inserts the hovering disk
                  into the current arrangement and onto the canvas
              """
              if event.name     == 'button_press_event' and \
                 event.button   == 1                    and \
                 event.dblclick == True                 and \
                 event.xdata    != None                 and \
                 event.ydata    != None                 and \
              (run.horse_obstacle_input_mode_p or run.fly_obstacle_input_mode_p or run.common_obstacle_input_mode_p) :

                  if run.horse_obstacle_input_mode_p:
                      fcol = 'crimson'
                      obstype = 'horseobs'

                  elif run.fly_obstacle_input_mode_p:
                       fcol = 'blue'
                       obstype = 'flyobs'

                  elif run.common_obstacle_input_mode_p:
                       fcol = 'green'
                       obstype = 'commonobs'

                  # Update the current disk list
                  run.obstacle_list.append( Obstacle( llcorner = [event.xdata,event.ydata], \
                                                width = run.hovering_obstacle_width, \
                                                height = run.hovering_obstacle_height, \
                                                figure      = fig, \
                                                axes_object = ax,
                                                obstype     = obstype  ,
                                                diskcolor   = fcol)  )

                  # Add representation of the disk appended to the canvas and show the updated count
                  ax.add_patch( run.obstacle_list[-1].mplPatch(fcol))

                  # Render the canvas
                  fig.canvas.draw()
```

```
        return _placeObstacle
```

◇

Fragment referenced in 3.

Obstacle sizes can be increased by using the `Shift` key and decreased using the `Ctrl` key.

⟨ *Implementation of* `wrapperResizeHoveringObstacle` 10 ⟩ ≡

```
def wrapperResizeHoveringObstacle(fig,ax,run):
        """ Wrapper for the call-back function _resizeHoveringObstacle
        """
        def _resizeHoveringObstacle(event):
            """ Each key-press increments or decrements by a fixed amount
            the radius of the hovering disk. Change the frozenset global config
            GC dictionary for changing the increment and decrement deltas corresponding
            to each key-press
            """

            # This used to be in the arguments to _resizeHoveringObstacle
            previous_patch=[]

            # Increase hovering disk radius
            if  event.key  == "shift":
                run.hovering_obstacle_height += 0.05
                run.hovering_obstacle_width += 0.05

                current_patch = mpl.patches.Rectangle((event.xdata,event.ydata),         \
                                                      width     = run.hovering_obstacle_width,  \
                                                      height    = run.hovering_obstacle_height, \
                                                      facecolor = 'black', \
                                                      alpha=0.2)
                previous_patch.append(current_patch)
                ax.add_patch(current_patch)

                fig.canvas.draw()

            elif event.key == "control" and \
                run.hovering_obstacle_width >= 2.0 * 0.05:

                run.hovering_obstacle_width  -= 0.05
                run.hovering_obstacle_height -= 0.05

                current_patch = mpl.patches.Rectangle((event.xdata,event.ydata),           \
                                                      width     = run.hovering_obstacle_width,  \
                                                      height    = run.hovering_obstacle_height, \
                                                      facecolor = 'black', \
                                                      alpha=0.2)
                previous_patch.append(current_patch)
                ax.add_patch(current_patch)

                fig.canvas.draw()

            while len(previous_patch) != 0:
                previous_patch.pop().remove()

        return _resizeHoveringObstacle
```

◇

Fragment referenced in 3.

Discretization of the Geometric Domain

Having placed all the obstacles in memory, we now discretize the underlying geometric domain. I will be using CGAL for performing Segment-Segment intersection tests while building visibiity graphs on any underlying grid. For the moment, during an initial implementation, I will assume that none of the sites are covered by any of the obstacles and no two obstacles intersect. The background grid points however, may lie inside the obstacles.

⟨ *Definition of method* makeHorseflyInputGraph 12 ⟩ ≡

```
def makeHorseflyInputGraph(self, fig, ax, background_grid_pts,
                           horse_graph_policy='visibility',
                           fly_graph_policy  ='visibility'):
    """
      background_grid_pts is a list of the points used in the background_grid
        (typically uniform grid points, or points distributed in the surrounding square
        according to some probability distribution. )

      There can be several types of policies for each of the horse and fly-graphs.
      The most basic one and the default, used is just the visibility graph on the set
      of points. A segment is in the visibility graph if it does not intersect any
      obstacle.
    """
    # Make a list of nbunch of nodes to be inserted into \verb|horsfly_input_graph|
    node_list = []
   num_non_obstacle_vertices = len(background_grid_pts)+len(run.sites) + 1 # the +1 is the initial position of horse and fly

    # Insert background_grid_pts
    for pt in background_grid_pts:
        node_list.append( {'coordinates': pt, 'point_type' : 'background'} )

    # Insert site points
    for site in run.sites:
        node_list.append({'coordinates':site, 'point_type': 'site'})

    # Insert inithorseposn if it has been provided
    if run.inithorseposn:
         node_list.append({'coordinates':run.inithorseposn, 'point_type':'inithorseposn'})

    # Insert obstacle vertices
    for obs in run.obstacle_list:
         for vertex in obs.getVertices():
             node_list.append({'coordinates':vertex, 'point_type':'obstacle_vertex'})

    # Give indexes to each node
    for node, idx in zip(node_list, range(len(node_list))):
         node['idx'] = idx

    utils_algo.print_list(node_list)

    # Make a list of ebunch of nodes to be inserted into \verb|horsfly_input_graph|
    from itertools import combinations
    potential_edge_list = list(combinations(node_list[:num_non_obstacle_vertices], 2))


    for pt_edge in potential_edge_list:

         [p,q] = pt_edge
         pcd = p['coordinates']
         qcd = q['coordinates']


         if np.linalg.norm(np.asarray(pcd)-np.asarray(qcd)) < 0.4:

             # IMPORTANT!! : A segment might intersect many obstacles.
             # 0. If a segment does not intersect any obstacles then the segment is included
             #    as an edge in the segment.
             # 1. If a segment intersects one or more obstacles precisely of type 'horseobs'
             #    then the edge is included and only a drone can travel along that edge
             # 2. If a segment intersects one or more obstacles precisely of type 'flyobs'
             #    then the edge is included and only a truck can travel along that edge
             # 3. If a segment intersects one or more obstacles precisely of type 'commonobs'
             #    then the segment is *NOT* included as an edge in the graph
             # 4. If a segment intersects one or more obstacles of more than one type, then
             #    the segment is *NOT* included as an edge in the graph
             # Besides this only those segments whose length is less than a certain given
             # tolerance is included in the graph. This will allow me to make the graph
             # denser and the domain more well-sampled. And more importantly the drawn
             # graph will be easily visualizable too.
```

Rendering the Discretized Graph

⟨ *Definition of method* `renderHorseflyInputGraph` 13 ⟩ ≡

```
def renderHorseflyInputGraph(self,fig,ax):
    pass
```

◇

Fragment referenced in 4.