Experimental Analyses of Heuristics for Horsefly-type Problems

Gaurish Telang

Contents

Part I

Overview

Chapter 1

Descriptions of Problems

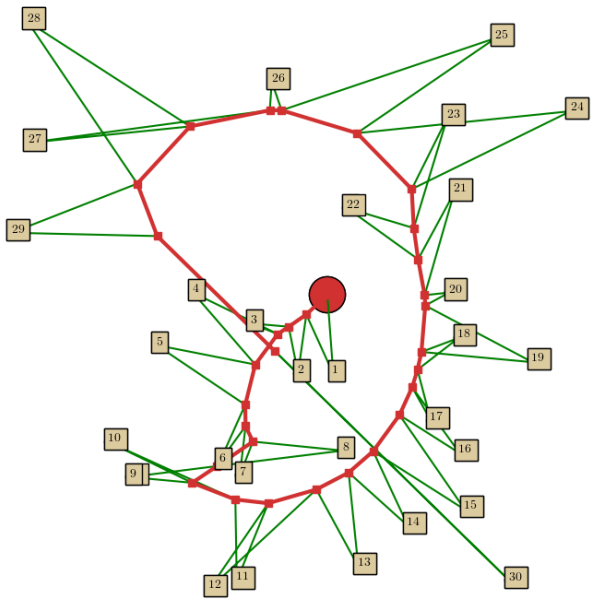

Figure 1.1: An Example of a classic Horsefly tour with $\varphi = 5$. The red dot indicates the initial position of the horse and fly, given as part of the input. The ordering of sites shown has been computed with a greedy algorithm which will be described later

The Horsefly problem is a generalization of the well-known Euclidean Traveling Salesman Problem. In the most basic version of the Horsefly problem (which we call **"Classic Horsefly"**), we are given a set of sites, the initial position of a truck(horse) with a drone(fly) mounted on top, and the speed of the drone-speed φ. [1] [2].

The goal is to compute a tour for both the truck and the drone to deliver package to sites as quickly as possible. For delivery, a drone must pick up a package from the truck, fly to the site and come back to the truck to pick up the next package for delivery to another site. [3]

[1] The speed of the truck is always assumed to be 1 in any of the problem variations we will be considering in this report.

[2] φ is also called the "speed ratio".

[3] The drone is assumed to be able to carry at most one package at a time

Both the truck and drone must coordinate their motions to minimize the time it takes for all the sites to get their packages. Figure 1.1 gives an example of such a tour computed using a greedy heuristic for $\varphi = 5$.

This suite of programs implement several experimental heuristics, to solve the above NP-hard problem and some of its variations approximately. In this short chapter, we give a description of the problem variations that we will be tackling. Each of the problems, has a corresponding chapter in Part 2, where these heuristics are described and implemented. We also give comparative analyses of their experimental performance on various problem instances.

Classic Horsefly This problem has already described in the introduction.

Segment Horsefly In this variation, the path of the truck is restricted to that of a segment, which we can consider without loss of generality to be $[0, 1]$. All sites, without loss of generality lie in the upper-half plane $\mathbb{R}^2_+$.

Fixed Route Horsefly This is the obvious generalization of Segment Horsefly, where the path which the truck is restricted to travel is a piece-wise linear polygonal path. [4] Both the initial position of the truck and the drone are given. The sites to be serviced are allowed to lie anywhere in $\mathbb{R}^2$. Two further variations are possible in this setting, one in which the truck is allowed reversals and the other in which it is not.

One Horse, Two Flies The truck is now equipped with two drones. Otherwise the setting, is exactly the same as in classic horsefly. Each drone can carry only one package at a time. The drones must fly back and forth between the truck and the sites to deliver the packages. We allow the possibility that both the drones can land at the same time and place on the truck to pick up their next package. [5]

Reverse Horsefly In this model, each site (not the truck!) is equipped with a drone, which fly *towards* the truck to pick up their packages. We need to coordinate the motion of the truck and drone so that the time it takes for the last drone to pick up its package (the "makespan") is minimized.

Bounded Distance Horsefly In most real-world scenarios, the drone will not be able to (or allowed to) go more than a certain distance R from the truck. Thus with the same settings as the classic horsefly, but with the added constraint of the drone and the truck never being more than a distance R from the truck, how would one compute the truck and drone paths to minimize the makespan of the deliveries?

[4] More generally, the truck will be restricted to travelling on a road network, which would typically be modelled as a graph embedded in the plane.

[5] In reality, one of the drones will have to wait for a small amount of time while the other is retrieving its package. In a more realisting model, we would need to take into account this "waiting time" too.

Watchman Horsefly In place of the TSP, we generalize the Watchman route problem here. [6] We are given as input a simple polygon and the initial position of a truck and a drone. The drone has a camera mounted on top which is assumed to have 360° vision. Both the truck and drone can move, but the drone can move at most euclidean distance [7] R from the truck.

We want every point in the polygon to be seen by the drone at least once. The goal is to minimize the time it takes for the drone to be able to see every point in the simple polygon. In other words, we want to minimize the time it takes for the drone (moving in coordinattion with the truck) to patrol the entire polygon.

[6] although abstractly, the Watchman route problem can be viewed as a kind of TSP

[7] The version where instead geodesic distance is considered is also interesting

Chapter 2

Installation and Use

To run these programs you will need to install Docker, an open-source containerization program that is easily installable on Windows 10[1], MacOS, and almost any GNU/Linux distribution. For a quick introduction to containerization, watch the first two minutes of https://youtu.be/_dfLOzuIg2o

The nice thing about Docker is that it makes it easy to run softwares on different OS'es portably and neatly side-steps the dependency hell problem (https://en.wikipedia.org/wiki/Dependency_hell.) The headache of installing different library dependencies correctly on different machines running different OS'es, is replaced **only** by learning how to install Docker and to set up an X-windows connection between the host OS and an instantiated container running GNU/Linux.

A. [*Get Docker*] For installation instrutions watch

GNU/Linux https://youtu.be/KCckWweNSrM

Windows https://youtu.be/ymlWt1MqURY

MacOS https://youtu.be/MU8HUVlJTEY

B. [*Download customized Ubuntu image*] `docker pull gtelang/ubuntu_customized` [2]

C. [*Clone repository*] `git clone gtelang/horseflies_literate.git`

D. [*Mount and Launch*]

For GNU/Linux Open up your favorite terminal emulator, such xterm and then

- Copy to clipboard the output of `xauth list`

[1] You might need to turn on virtualization explicitly in your BIOS, after installing Docker as I needed to while setting Docker up on Windows. Here is a snapshot of an image when turning on Intel's virtualization technology through the BIOS: https://images.techhive.com/images/article/2015/09/virtualbox_vt-x_amd-v_error04_phoenix-100612961-large.idge.jpg

[2] The customized Ubuntu image is approximately 7 GB which contains all the libraries (e.g. CGAL, VTK, numpy, and matplotlib) that I typically use to run my research codes portably.On my home internet connection downloading this Ubuntu-image typically takes about 5 minutes.

- `cd horseflies_literate`
- `docker run -it --name horsefly_container --net=host -e DISPLAY -v /tmp/.X11-unix -`
- `cd horseflies_mnt`
- `xauth add` *<paste-from-clipboard>*

For Windows I had to follow the instructions in https://dev.to/darksmile92/run-gui-app-in-linux-Docker-container-on-windows-host-4kde to be able to run graphical user applications

E. [*Run experiments*] If you want to run all the experiments as described in the paper again to reproduce the reported results on your machine, then run [3],
`python main.py --run-all-experiments`.

If you want to run a specific experiment, then run
`python main.py --run-experiment <experiment-name>`.

See Index for a list of all the experiments.

F. [*Test algorithms interactively*] If you want to test the algorithms in interactive mode (where you get to select the problem-type, mouse-in the sites on a canvas, set the initial position of the truck and drone and set φ), run `python main.py --<problem-name>`. The list of problems are the same as that given in the previous chapter. The problem name consists of all lower-case letters with spaces replaced by hyphens.

Thus for instance "Watchman Horsefly" becomes `watchman-horsefly` and "One Horse Two Flies" becomes `one-horse-two-flies`.

To interactively experiment with different algorithms for, say, the Watchman Horsefly problem , type at the terminal `python main.py --watchman-horsefly`

If you want to delete the Ubuntu image and any associated containers run the command [4]

```
docker rm -f horsefly_container; docker rmi -f ubuntu_customized
```

That's it! Happy horseflying!

[3] Allowing, of course, for differences between your machine's CPU and mine when it comes to reporting absolute running time

[4] the ubuntu image is 7GB afterall!

Sunday 16th December, 2018 01:47

Part II

Programs

Chapter 3

Overview of the Code Base

NOTE: The style of presentation in this chapter has been adapted from Chapter 2 of the Nuweb reference manual http://nuweb.sourceforge.net/nuweb.pdf

Almost all of the code has been written in Python 2.7 and tested using the standard CPython implementation of the language. In some cases, calls will be made to external C++ libraries (mostly CGAL and VTK) using SWIG (http://www.swig.org/). This is either for speeding up a slow routine or to use a function that is not available in any existing Python package.

There are three principal directories

`webs/` This contains the source code for the entire project written in the nuweb format along with documents (mostly images) needed during the compilation of the LaTeX files which will be extracted from the `.web` files.

`src/` This contains the source code for the entire project "tangled" (i.e. extracted) from the `.web` files.

`tex/` This contains the monolithic `horseflies.tex` extracted from the `.web` files and a bunch of other supporing LaTeX files. It also contains the final compiled `horseflies.pdf` (the current document) which contains the documentation of the project, interwoven with code-chunks and cross-references between them along with the experimental results.

The files in `src` and `tex` should not be touched. Any editing required should be done to the `.web` files, which should then be weaved and tangled using the script `weave-tangle.sh` in the `webs` directory.

The Main Files

3.1.1 The file `main.py` is the entry-point for running code. This file takes care of reading command-line arguments and accordingly executing the interactive or experimental sections of the code for the appropriate problem. This file is in the top-level folder.

3.1.2 Each of the files with prefix `algos-*` contain implmentations of algorithms for one specific problem. Thus `algos-watchman-horsefly.py` contains algorithms for approximately solving the Watchman Horsefly problem.

All such files are in the directory `src/lib/`

3.1.3 Similarly, each of the files with prefix `expt-*` contain code for testing hypotheses regarding a problem, generating counter-examples or comparing the experimental performance of the algorithm implementations for each of the problems. Thus `expt-watchman-horsefly.py` contains code for performing experiments related to the Watchman Horsefly problem.

All such files are in the directory `src/expt/`

Support Files

3.2.1 These files contain common utility functions that will be useful for manipulating data-structures, common plotting and graphics routines for all horsefly-type problems. All such files have the prefix `utils-*`

All such files are in the directory `src/lib/`

3.2.2 To automate testing of code during implementations, tests for various routines across the entire code-base have been written in files with prefix `test-*`.

Each of the main files have a corresponding test file. Tests for functions in the support files and experimental files have all been implemented in the files `test-utilities.py` and `test-experiments.py` respectively.

All such files are in the directory `src/test/`

Chapter 4

Classic Horsefly

Chapter 5

Segment Horsefly

Chapter 6

Fixed Route Horsefly

Chapter 7

One Horse, Two Flies

Chapter 8

Reverse Horsefly

Chapter 9

Watchman Horsefly

Appendices

Appendix A

Index of Files

Appendix B

Index of Fragments

None.

Appendix C

Index of Identifiers

Appendix D

Man-page for `main.py`