Chapter 8

Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

8.1 Problem statement

Scheduling the route of a delivery truck is a well-studied problem. For instance, minimizing the total distance that the truck has to travel corresponds to the *Traveling Salesman Problem (TSP)*. However, in the case of *Amazon delivery truck* scheduling the problem has some new aspects:

- A customer is promised a window of days when delivery can take place. Thus, the truck can split the list of places into sublists, with a shorter total distance than going through the list in one sweep.
- Except that *Amazon prime* customers need their deliveries guaranteed the next day.

8.2 Coding up the basics

Before we try finding the best route, let's put the basics in place to have any sort of route at all.

8.2.1 Address list

You probably need a class `Address` that describes the location of a house where a delivery has to be made.

- For simplicity, let give a house (i, j) coordinates.
- We probably need a `distance` function between two addresses. We can either assume that we can travel in a straight line between two houses, or that the city is build on a grid, and you can apply the so-called *Manhattan distance*.
- The address may also require a field recording the last possible delivery date.

Exercise 8.1. Code a class `Address` with the above functionality, and test it.

This document is available on studocu
Downloaded by manuel vicenty (vicentymanuel4@gmail.com)

Downloaded by manuel vicenty (vicentymanuel4@gmail.com)

Code:

```
    Address one(1.,1.),
      two(2.,2.);
    cerr << "Distance: "
         << one.distance(two)
         << '\n';
```

Output

[amazon] address:

```
Address
Distance: 1.41421
.. address
Address 1 should be closest to
    the depot. Check: 1

Route from depot to depot:
    (0,0) (2,0) (1,0) (3,0)
    (0,0)
 has length 8: 8
Greedy scheduling:  (0,0) (1,0)
    (2,0) (3,0) (0,0)
 should have length 6: 6

Square5
Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
.. square5

Original list:  (0,0) (-2,0)
    (-1,0) (1,0) (2,0) (0,0)
 length=8
flip middle two addresses:
    (0,0) (-2,0) (1,0) (-1,0)
    (2,0) (0,0)
 length=12
better:  (0,0) (1,0) (-2,0)
    (-1,0) (2,0) (0,0)
 length=10

Hundred houses
Route in order has length
    25852.6
TSP based on mere listing has
    length: 2751.99 over naive
    25852.6
Single route has length: 2078.43
 .. new route accepted with
    length 2076.65
Final route has length 2076.65
    over initial 2078.43
TSP route has length 1899.4
    over initial 2078.43

Two routes
Route1:  (0,0) (2,0) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (3,1) (2,1)
    (1,2) (1,3) (0,0)
total length 19.6251
start with 9.88635,9.73877
Pass 0
.. down to 9.81256,8.57649
Pass 1
Pass 2
Pass 3
Pass 4
TSP Route1:  (0,0) (3,1) (3,2)
    (2,3) (0,2) (0,0)
route2:  (0,0) (2,0) (2,1)
    (1,2) (1,3) (0,0)
total length 18.389
```


This document is available on studocu
Downloaded by manuel vicenty (vicentymanuel4@gmail.com)

Next we need a class `AddressList` that contains a list of addresses.

Exercise 8.2. Implement a class `AddressList`; it probably needs the following methods:

- `add_address` for constructing the list;
- `length` to give the distance one has to travel to visit all addresses in order;
- `index_closest_to` that gives you the address on the list closests to another address, presumably not on the list.

8.2.2 Add a depot

Next, we model the fact that the route needs to start and end at the depot, which we put arbitrarily at coordinates $(0, 0)$. We could construct an `AddressList` that has the depot as first and last element, but that may run into problems:

- If we reorder the list to minimize the driving distance, the first and last elements may not stay in place.
- We may want elements of a list to be unique: having an address twice means two deliveries at the same address, so the `add_address` method would check that an address is not already in the list.

We can solve this by making a class `Route`, which inherits from `AddressList`, but the methods of which leave the first and last element of the list in place.

8.2.3 Greedy construction of a route

Next we need to construct a route. Rather than solving the full TSP, we start by employing a *greedy search* strategy:

> Given a point, find the next point by some local optimality test, such as shortest distance. Never look back to revisit the route you have constructed so far.

Such a strategy is likely to give an improvement, but most likely will not give the optimal route.

Let's write a method

```
Route::Route greedy_route();
```

that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

Exercise 8.3. Write the `greedy_route` method for the `AddressList` class.

1. Assume that the route starts at the depot, which is located at $(0, 0)$. Then incrementally construct a new list by:
2. Maintain an `Address` variable `we_are_here` of the current location;
3. repeatedly find the address closest to `we_are_here`.

Extend this to a method for the `Route` class by working on the subvector that does not contain the final element.

Test it on this example:

Downloaded by manuel vicenty (vicentymanuel4@gmail.com)

Code:

```
Route deliveries;
deliveries.add_address( Address(0,5)
);
deliveries.add_address( Address(5,0)
);
deliveries.add_address( Address(5,5)
);
cerr << "Travel in order: " <<
deliveries.length() << '\n';
assert( deliveries.size()==5 );
auto route =
deliveries.greedy_route();
assert( route.size()==5 );
auto len = route.length();
cerr << "Square route: " <<
route.as_string()
     << "\n has length " << len <<
'\n';
```

Output
[amazon] square5:

```
Travel in order: 24.1421
Square route:  (0,0) (0,5)
    (5,5) (5,0) (0,0)
 has length 20
```

Reorganizing a list can be done in a number of ways.

- First of all, you can try to make the changes in place. This runs into the objection that maybe you want to save the original list; also, while swapping two elements can be done with the `insert` and `erase` methods, more complicated operations are tricky.
- Alternatively, you can incrementally construct a new list. Now the main problem is to keep track of which elements of the original have been processed. You could do this by giving each adress a boolean field `done`, but you could also make a copy of the input list, and remove the elements that have been processed. For this, study the `erase` method for `vector` objects.

8.3 Optimizing the route

The above suggestion of each time finding the closest address is known as a *greedy search* strategy. It does not give you the optimal solution of the TSP. Finding the optimal solution of the TSP is hard to program – you could do it recursively – and takes a lot of time as the number of addresses grows. In fact, the TSP is probably the most famous of the class of *NP-hard* problems, which are generally believed to have a running time that grows faster than polynomial in the problem size.

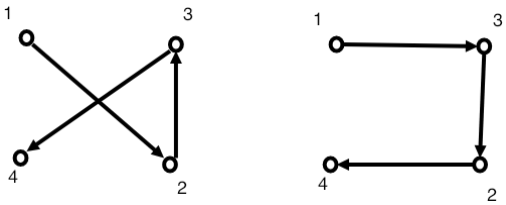

Figure 8.1: Illustration of the 'opt2' idea of reversing part of a path

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [8], is based on the *opt2* idea: if you have a path that 'crosses itself', you can make it shorter by reversing

This document is available on studocu
Downloaded by manuel vicenty (vicentymanuel4@gmail.com)

part of it. Figure 8.1Illustration of the 'opt2' idea of reversing part of a pathfigure.8.1 shows that the path $1 - -2 - -3 - -4$ can be made shorter by reversing part of it, giving $1 - -3 - -2 - -4$. Since recognizing where a path crosses itself can be hard, or even impossible for graphs that don't have Cartesian coordinates associated, we adopt a scheme:

```
for all nodes m<n on the path [1..N]:
  make a new route from
    [1..m-1] + [m--n].reversed + [n+1..N]
  if the new route is shorter, keep it
```

Exercise 8.4. Code the opt2 heuristic: write a method to reverse part of the route, and write the loop that tries this with multiple starting and ending points. Try it out on some simple test cases to convince you that your code works as intended.

Exercise 8.5. What is the runtime complexity of this heuristic solution?

Exercise 8.6. Earlier you had programmed the greedy heuristic. Compare the improvement you get from the opt2 heuristic, starting both with the given list of addresses, and with a greedy traversal of it.

8.4 Multiple trucks

If we introduce multiple delivery trucks, we get the 'Multiple Traveling Salesman Problem' [3]. With this we can module both the cases of multiple trucks being out on delivery on the same day, or one truck spreading deliveries over multiple days. For now we don't distinguish between the two.

The first question is how to divide up the addresses.

1. We could split the list in two, using some geometric test. This is a good model for the case where multiple trucks are out on the same day. However, if we use this as a model for the same truck being out on multiple days, we are missing the fact that new addresses can be added on the first day, messing up the neatly separated routes.
2. Thus it may in fact be reasonable to assume that all trucks get an essentially random list of addresses.

Can we extend the opt2 heuristic to the case of multiple paths? For inspiration take a look at figure 8.2Extending the 'opt2' idea to multiple pathsfigure.8.2: instead of modifying one path, we could switch bits out bits between one path and another. When you write the code, take into account that the other path may be running backwards! This means that based on split points in the first and second path you know have four resulting modified paths to consider.

Exercise 8.7. Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 8.3Multiple paths test casefigure.8.3.

You have quite a bit of freedom here:

- The start points of the two segments should be chosen independently;
- the lengths can be chosen independently, but need not; and finally
- each segment can be reversed.

Downloaded by manuel vicenty (vicentymanuel4@gmail.com)

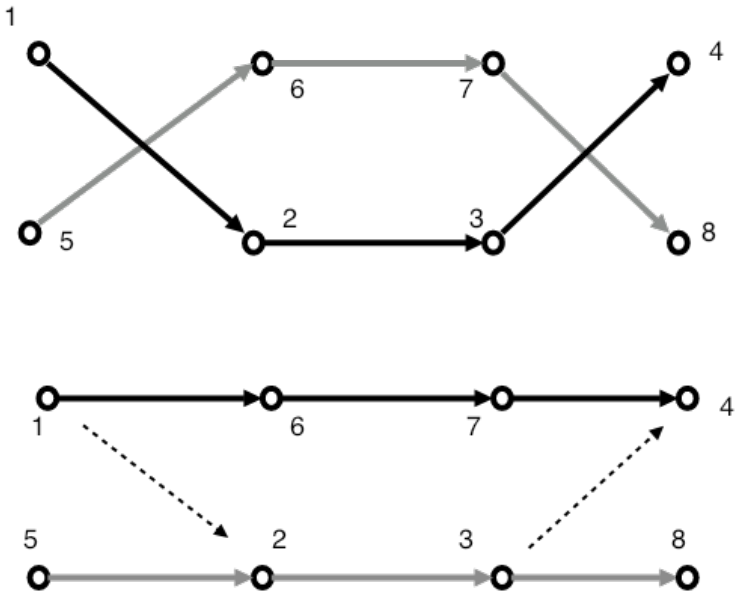

Figure 8.2: Extending the 'opt2' idea to multiple paths

More flexibility also means a longer runtime of your program. Does it pay off? Do some tests and report results.

Based on the above description there will be a lot of code duplication. Make sure to introduce functions and methods for various operations.

8.5 Amazon prime

In section 8.4Multiple truckssection.8.4 you made the assumption that it doesn't matter on what day a package is delivered. This changes with *Amazon prime*, where a package has to be delivered guaranteed on the next day.

Exercise 8.8. Explore a scenario where there are two trucks, and each have a number of addresses that can not be exchanged with the other route. How much longer is the total distance? Experiment with the ratio of prime to non-prime addresses.

8.6 Dynamicism

So far we have assumed that the list of addresses to be delivered to is given. This is of course not true: new deliveries will need to be scheduled continuously.

Exercise 8.9. Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

This document is available on studocu
Downloaded by manuel vicenty (vicentymanuel4@gmail.com)

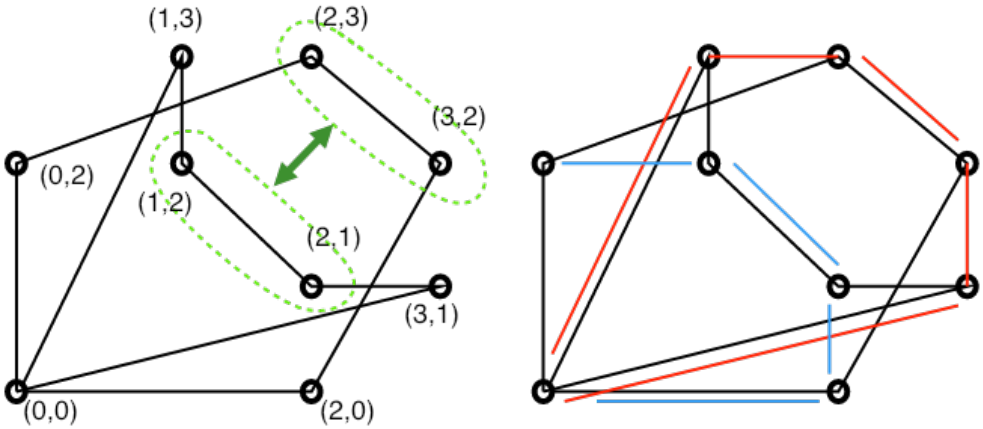

Figure 8.3: Multiple paths test case

8.7 Ethics

People sometimes criticize Amazon's labor policies, including regarding its drivers. Can you make any observations from your simulations in this respect?

Downloaded by manuel vicenty (vicentymanuel4@gmail.com)