Programming project: Amazon delivery truck scheduling

Victor Eijkhout

2020
Abstract

Delivery trucks are all over town, getting as many packages to people as quickly as possible. Is it possible to find the best route through town? This is an important question for many companies, such as Amazon. (The US Postal Service solves a different problem.) In this simple form this is the Traveling Salesman Problem (TSP), which has been studied extensively. But Amazon has some liberties: packages can be spread out over days (unless you have Amazon Prime), and there are multiple trucks to divide the area over.

This project walks a student through a heuristic solution of the TSP and the ‘multiple TSP’. With the full implementation, students can explore some scenarios.

The project description explicitly targets an object-oriented formulation; this project is much harder in a non-OO language.

This project is preferably done by two students in close collaboration; it will take about two weeks at the end of a first or second semester programming course.
<table>
<thead>
<tr>
<th>Summary</th>
<th>Model the ‘Multiple Traveling Salesman Problem.’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topics</td>
<td>Recursion, arrays, classes</td>
</tr>
<tr>
<td>Audience</td>
<td>Undergraduate or AP high school</td>
</tr>
<tr>
<td>Difficulty</td>
<td>Moderate to high</td>
</tr>
<tr>
<td>Strengths</td>
<td>Appealing, opportunity for experimentation</td>
</tr>
<tr>
<td>Weaknesses</td>
<td>Requires many pieces to be in place before the code does something useful.</td>
</tr>
<tr>
<td></td>
<td>No graphics output, so the interpretation of output requires some imagination</td>
</tr>
<tr>
<td>Dependencies</td>
<td>No specific background in CS (such as the TSP) is needed. Familiarity with the notions of heuristics and search is welcome.</td>
</tr>
<tr>
<td>Variants</td>
<td>Application slight different problem statements, such as HEB (big supermarket chain) curb-side delivery shoppers.</td>
</tr>
</tbody>
</table>
Chapter 1

Amazon delivery truck scheduling

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

1.1 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.

1.2 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.

1.2.1 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 1.1.** Code a class `Address` with the above functionality, and test it.
1. Amazon delivery truck scheduling
1.2. Coding up the basics

Code:

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

Output

[amazon] address:

Address

Distance: 1.41421
.. address

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) (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=10
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)
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

Two routes
Total route has greedy length 2677.46 and TSP length 2474.37
Greedy routes have length 1817.21, 1932.43
start with 1817.21,1932.43
Pass 0
.. down to 1813.43,1926.55
.. down to 1811.74,1924.52
.. down to 1803.16,1893.5
.. down to 1786.61,1890.77
.. down to 1784.83,1888.22
.. down to 1779.53,1885.13
.. down to 1779.1,1884.69
.. down to 1774.08,1849.38
.. down to 1763.01,1849.15
.. down to 1759.65,1835.71
.. down to 1750.57,1830.57
.. down to 1748.53,1827.73
.. down to 1747.9,1814.22
.. down to 1744.81,1808.56
.. down to 1744.65,1806.8
.. down to 1739.43,1799.36
.. down to 1738.92,1770.71
.. down to 1737.68,1769.23
.. down to 1733.91,1752.42
.. down to 1729.19,1749.4
.. down to 1723.37,1748.81
.. down to 1720.09,1738.53
.. down to 1718.82,1706.81
.. down to 1717.81,1698.53
.. down to 1713.67,1697.38
.. down to 1712.85,1689.72
.. down to 1699.47,1683.92
.. down to 1693.73,1670.63
.. down to 1691.52,1670
.. down to 1690.75,1669.64
.. down to 1685.12,1647.72
.. down to 1681.07,1647.23
.. down to 1675.52,1645.16
.. down to 1651.96,1630.78
.. down to 1640.85,1627.69
.. down to 1637.9,1625.78
.. down to 1630.85,1620.57
.. down to 1623.2,1614.22
.. down to 1613.79,1608.56
.. down to 1609.41,1606.8
.. down to 1603.43,1601.62
.. down to 1599.41,1576.3
.. down to 1593.92,1570.71
.. down to 1589.69,1567.55
.. down to 1584.93,1564.93
.. down to 1579.65,1561.31
.. down to 1574.08,1558.38
.. down to 1563.01,1549.15
.. down to 1559.65,1535.71
.. down to 1550.57,1530.57
.. down to 1548.53,1527.73
.. down to 1547.9,1514.22
.. down to 1544.81,1508.56
Next we need a class \textit{AddressList} that contains a list of addresses.

\begin{exercise}
Implement a class \textit{AddressList}; it probably needs the following methods:
\begin{itemize}
  \item \textit{add_address} for constructing the list;
  \item \textit{length} to give the distance one has to travel to visit all addresses in order;
  \item \textit{index\_closest\_to} that gives you the address on the list closest to another address, presumably not on the list.
\end{itemize}
\end{exercise}

\subsection{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 \textit{AddressList} that has the depot as first and last element, but that may run into problems:
\begin{itemize}
  \item If we reorder the list to minimize the driving distance, the first and last elements may not stay in place.
  \item We may want elements of a list to be unique: having an address twice means two deliveries at the same address, so the \textit{add\_address} method would check that an address is not already in the list.
\end{itemize}
We can solve this by making a class \textit{Route}, which inherits from \textit{AddressList}, but the methods of which leave the first and last element of the list in place.

\subsection{Greedy construction of a route}

Next we need to construct a route. Rather than solving the full TSP, we start by employing a \textit{greedy search} strategy:
\begin{quote}
  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.
\end{quote}
Such a strategy is likely to give an improvement, but most likely will not give the optimal route.

Let’s write a method
\begin{verbatim}
\| Route::Route greedy\_route();
\end{verbatim}
that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

\begin{exercise}
Write the \textit{greedy\_route} method for the \textit{AddressList} class.
\begin{enumerate}
  \item Assume that the route starts at the depot, which is located at \((0, 0)\). Then incrementally construct a new list by:
  \item Maintain an \textit{Address} variable \textit{we\_are\_here} of the current location;
  \item repeatedly find the address closest to \textit{we\_are\_here}.
\end{enumerate}
Extend this to a method for the \textit{Route} class by working on the subvector that does not contain the final element.
Test it on this example:
\end{exercise}
1.3 Optimizing the route

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 address 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.

1.3 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 1.1: Illustration of the ‘opt2’ idea of reversing part of a path](image)

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [2], is based on the opt2 idea: if you have a path that ‘crosses itself’, you can make it shorter by reversing part of it. Figure 1.1 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:

```cpp
for all nodes m<n on the path [1..N]:
    make a new route from
    [1..m-1] + [m--n].reversed + [n+1..N]
```

Victor Eijkhout
1. Amazon delivery truck scheduling

if the new route is shorter, keep it

Exercise 1.4. 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 1.5. What is the runtime complexity of this heuristic solution?

Exercise 1.6. 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.

1.4 Multiple trucks

If we introduce multiple delivery trucks, we get the ‘Multiple Traveling Salesman Problem’ [1]. With this we can model 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 1.2: 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 1.7. Write a function that optimizes two paths simultaneously using the multi-path version of the opt2 heuristic. For a test case, see figure 1.3.

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

In section 1.4 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 1.8. 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.

1.6 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 1.9. Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.
1. Amazon delivery truck scheduling

Figure 1.3: Multiple paths test case
Chapter 2

Style guide for project submissions

_The purpose of computing is insight, not numbers. (Richard Hamming)_

Your project writeup is at least as important as the code. Here are some common-sense guidelines for a good writeup. However, not all parts may apply to your project. Use your good judgement.

**Style** First of all, observe correct spelling and grammar. Use full sentences.

**Completeness** Your writeup needs to have the same elements as a good paper:
- Title and author, including EID.
- A one-paragraph abstract.
- A bibliography at the end.

**Introduction** The reader of your document need not be familiar with the project description, or even the problem it addresses. Indicate what the problem is, give theoretical background if appropriate, possibly sketch a historic background, and describe in global terms how you set out to solve the problem, as well as your findings.

**Code** Your report should describe in a global manner the algorithms you developed, and you should include relevant code snippets. If you want to include full listings, relegate that to an appendix: code snippets should only be used to illustrate especially salient points.

Do not use screen shots of your code: at the very least use a `verbatim` environment, but using the `listings` package (used in this book) is very much recommended.

**Results and discussion** Present tables and/or graphs when appropriate, but also include verbiage to explain what conclusions can be drawn from them.

You can also discuss possible extensions of your work to cases not covered.

**Summary** Summarize your work and findings.
2. Style guide for project submissions
Chapter 3

Bibliography