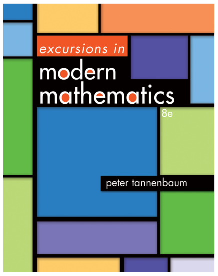


Chapter 6: The Mathematics of Touring

6.3 The Brute-Force Algorithm

Algorithms for Solving TSP's

In this section we will look at the two strategies we informally developed in connection with Willy's sales trips and recast them in the language of algorithms. The "exhaustive search" strategy can be formalized into an algorithm generally known as the **brute-force algorithm**; the "go cheap" strategy can be formalized into an algorithm known as the **nearest-neighbor algorithm**.

ALGORITHM 1: THE BRUTE FORCE ALGORITHM

- **Step 1.** Make a list of *all* the possible Hamilton circuits of the graph. Each of these circuits represents a tour of the vertices of the graph.
- **Step 2.** For each tour calculate its weight (i.e., add the weights of all the edges in the circuit).
- **Step 3.** Choose an *optimal* tour (there is always more than one optimal tour to choose from!).

Pros and Cons

The positive aspect of the brute-force algorithm is that it is an optimal algorithm. (An **optimal algorithm** is an algorithm that, when correctly implemented, is guaranteed to produce an optimal solution.) In the case of the brute-force algorithm, we know we are getting an optimal solution because we are choosing from among all possible tours.

Pros and Cons

The negative aspect of the brute-force algorithm is the amount of effort that goes into implementing the algorithm, which is (roughly) proportional to the number of Hamilton circuits in the graph. As we first saw in Table 6-4 in the text, as the number of vertices grows just a little, the number of Hamilton circuits in a complete graph grows at an incredibly fast rate.

Practical Terms of the Pros and Cons

What does this mean in practical terms? Let's start with human computation. To solve a TSP for a graph with 10 vertices (as real-life problems go, that's puny), the brute-force algorithm requires checking 362,880 Hamilton circuits. To do this by hand, even for a fast and clever human, it would take over 1000 hours. Thus, $N = 10$ at we are already beyond the limit of what can be considered reasonable human effort.

Using a Computer

Even with the world's best technology on our side, we very quickly reach the point beyond which using the brute-force algorithm is completely unrealistic.

TABLE 6-6

N	SUPERHERO computation time
20	2 minutes
21	40 minutes
22	14 hours
23	13 days
24	10 months
25	20 years
26	500 years
27	13,000 years
28	350,000 years
29	9.8 million years
30	284 million years

 PEARSON 6.3-7

Using a Computer

The brute-force algorithm is a classic example of what is formally known as an **inefficient algorithm**–an algorithm for which the number of steps needed to carry it out grows disproportionately with the size of the problem. The trouble with inefficient algorithms is that they are of limited practical use–they can realistically be carried out only when the problem is small.

 PEARSON 6.3-8

Pros and Cons - Nearest Neighbor

We hop from vertex to vertex using a simple criterion: Choose the next available "nearest" vertex and go for it. Let's call the process of checking among the available vertices and finding the nearest one a single computation. Then, for a TSP with $N = 5$ we need to perform 5 computations. What happens when we double, the number of vertices to $N = 10$? We now have to perform 10 computations. For $N = 30$, we perform 30 computations.

 PEARSON 6.3-9

Pros and Cons - Nearest Neighbor

We can summarize the above observations by saying that the nearest-neighbor algorithm is an *efficient algorithm*. Roughly speaking, an **efficient algorithm** is an algorithm for which the amount of computational effort required to implement the algorithm grows in some reasonable proportion with the size of the input to the problem.

 PEARSON 6.3-10

Pros and Cons - Nearest Neighbor

The main problem with the nearest-neighbor algorithm is that it is *not an optimal algorithm*. In Example 6.1, the **nearest-neighbor** tour had a cost of \$773, whereas the *optimal tour* had a cost of \$676. In absolute terms the nearest-neighbor tour is off by \$97 (this is called the absolute error). A better way to describe how far "off" this tour is from the optimal tour is to use the concept of *relative error*. In this example the absolute error is \$97 out of \$676, giving a relative error of \$97/\$676 = 0.1434911 ≈ 14.35%.

 PEARSON 6.3-11

In general, for any tour that might be proposed as a "solution" to a TSP, we can find its **relative error** ε as follows:

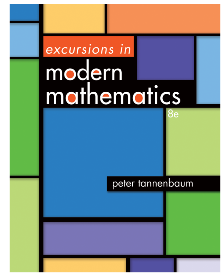
RELATIVE ERROR OF A TOUR (e)

$$\varepsilon = \frac{(\text{cost of tour} - \text{cost of optimal tour})}{\text{cost of optimal tour}}$$

 PEARSON 6.3-12

Relative Error - Good or Not Good

It is customary to express the relative error as a percentage (usually rounded to two decimal places). By using the notion of relative error, we can characterize the optimal tours as those with relative error of 0%. All other tours give "approximate solutions," the relative merits of which we can judge by their respective relative errors: tours with "small" relative errors are good, and tours with "large" relative errors are not good.



Chapter 6: The Mathematics of Touring

6.4 The Nearest-Neighbor and Repetitive Nearest-Neighbor Algorithms

ALGORITHM 2: THE NEAREST NEIGHBOR ALGORITHM

- **Start:** Start at the designated starting vertex. If there is no designated starting vertex pick any vertex.
- **First step:** From the starting vertex go to its nearest neighbor (i.e., the vertex for which the corresponding edge has the smallest weight).

ALGORITHM 2: THE NEAREST NEIGHBOR ALGORITHM

- **Middle steps:** From each vertex go to its nearest neighbor, choosing only among the vertices that haven't been yet visited. (If there is more than one nearest neighbor choose among them at random.) Keep doing this until all the vertices have been visited.
- **Last step:** From the last vertex return to the starting vertex.

Repetitive Nearest-Neighbor Algorithm

As one might guess, the repetitive nearest-neighbor algorithm is a variation of the nearest-neighbor algorithm in which we repeat several times the entire nearest- neighbor circuit-building process. Why would we want to do this? The reason is that the nearest-neighbor tour depends on the choice of the starting vertex. If we change the starting vertex, the nearest-neighbor tour is likely to be different, and, if we are lucky, better.

Repetitive Nearest-Neighbor Algorithm

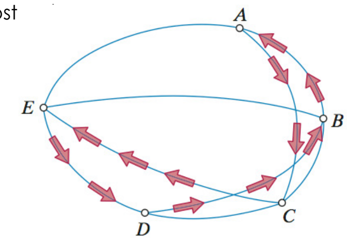
Since finding a nearest-neighbor tour is an efficient process, it is not unreasonable to repeat the process several times, each time starting at a different vertex of the graph. In this way, we can obtain several different "nearest-neighbor solutions," from which we can then pick the best.

Repetitive Nearest-Neighbor Algorithm

But what do we do with a tour that starts somewhere other than the vertex we really want to start at? That's not a problem. Remember that once we have a circuit, we can start the circuit anywhere we want. In fact, in an abstract sense, a circuit has no starting or ending point.

Example 6.5 A Tour of Five Cities: Part 3

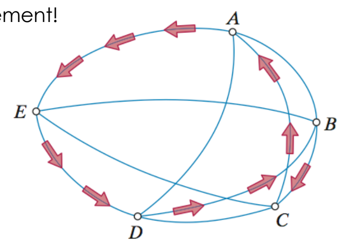
In Example 6.1, we computed the nearest-neighbor tour with *A* as the starting vertex, and we got *A*, *C*, *E*, *D*, *B*, *A* with a total cost



Example 6.5 A Tour of Five Cities: Part 3

But if we use *B* as the starting vertex, the nearest-neighbor tour takes us from *B* to *C*, then to *A*, *E*, *D*, and back to *B*, with a total cost of $722. Well, that is certainly an improvement!

As for Willy–who must start and end his trip at *A*–this very same tour would take the form *A*, *E*, *D*, *B*, *C*, *A*.

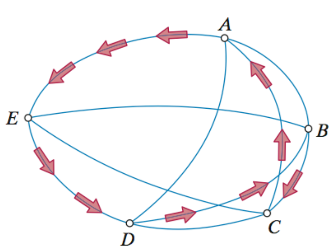


Example 6.5 A Tour of Five Cities: Part 3

The process is once again repeated using *C*, *D*, and *E* as the starting vertices. When the starting point is *C*, we get the nearest-neighbor tour *C*, *A*, *E*, *D*, *B*, *C* (total cost is $722); when the starting point is *D*, we get the nearest-neighbor tour *D*, *B*, *C*, *A*, *E*, *D* (total cost is $722); and when the starting point is *E*, we get the nearest-neighbor tour *E*, *C*, *A*, *D*, *B*, *E* (total cost is $741).

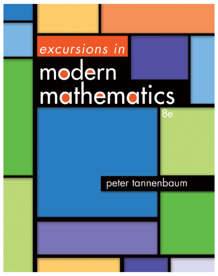
Example 6.5 A Tour of Five Cities: Part 3

Among all the nearest-neighbor tours we pick the cheapest one – *A*, *E*, *D*, *B*, *C*, *A* – with a cost of $722.



ALGORITHM 3: THE REPETITIVE NEAREST NEIGHBOR ALGORITHM

- Let *X* be any vertex. Find the nearest-neighbor tour with *X* as the starting vertex, and calculate the cost of this tour.
- Repeat the process with each of the other vertices of the graph as the starting vertex.
- Of the nearest-neighbor tours thus obtained, choose one with least cost. If necessary, rewrite the tour so that it starts at the designated starting vertex. We will call this tour the repetitive nearest-neighbor tour.



Chapter 6: The Mathematics of Touring

6.5 The Cheapest-Link Algorithm

Cheapest-Link Algorithm

The idea behind the **cheapest-link algorithm** is to piece together a tour by picking the separate "links" (i.e., legs) of the tour on the basis of cost. It doesn't matter if in the intermediate stages the "links" are all over the place – if we are careful at the end, they will all come together and form a tour. This is how it works: Look at the entire graph and choose the cheapest edge of the graph, wherever that edge may be. Once this is done, choose the next cheapest edge of the graph, wherever that edge may be.

 PEARSON

Cheapest-Link Algorithm

(Don't worry if that edge is not adjacent to the first edge.) Continue this way, each time choosing the cheapest edge available but following two rules:

(1) Do not allow a circuit to form except at the very end, and

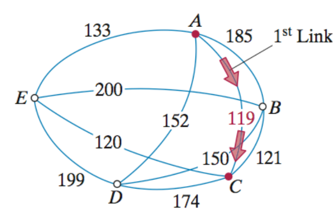
(2) Do not allow three edges to come together at a vertex.

A violation of either of these two rules will prevent forming a Hamilton circuit. Conversely, following these two rules guarantees that the end result will be a Hamilton circuit.

 PEARSON

Example 6.6 A Tour of Five Cities: Part 4

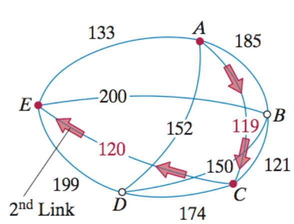
Among all the edges of the graph, the "cheapest link" is edge *AC*, with a cost of $119. For the purposes of recordkeeping, we will tag this edge as taken by marking it in red as shown.



 PEARSON

Example 6.6 A Tour of Five Cities: Part 4

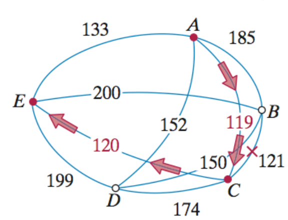
The next step is to scan the entire graph again and choose the cheapest link available, in this case edge *CE* with a cost of $120. Again, we mark it in red, to indicate it is taken.



 PEARSON

Example 6.6 A Tour of Five Cities: Part 4

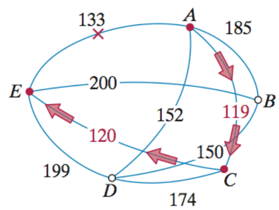
The next cheapest link available is edge *BC* ($121), but we should not choose *BC*–we would have three edges coming out of vertex *C* and this would prevent us from forming a circuit. This time we put a "do not use" mark on edge *BC*.



 PEARSON

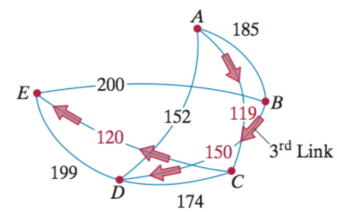
Example 6.6 A Tour of Five Cities: Part 4

The next cheapest link available is *AE* ($133). But we can't take *AE* either–the vertices *A*, *C*, and *E* would form a small circuit, making it impossible to form a Hamilton circuit at the end. So we put a "do not use" mark on *AE*.



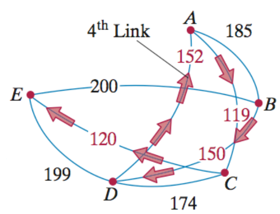
Example 6.6 A Tour of Five Cities: Part 4

The next cheapest link available is *BD* ($150). Choosing *BD* would not violate either of the two rules, so we can add it to our budding circuit and mark it in red.



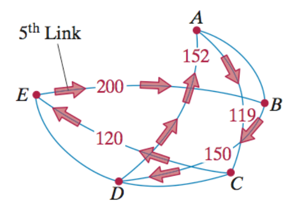
Example 6.6 A Tour of Five Cities: Part 4

The next cheapest link available is *AD* ($152), and it works just fine.



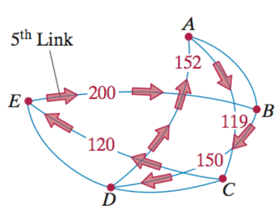
Example 6.6 A Tour of Five Cities: Part 4

At this point, we have only one way to close up the Hamilton circuit, edge *BE*, as shown.



Example 6.6 A Tour of Five Cities: Part 4

The cheapest-link tour shown could have any starting vertex we want. Since Willy lives at *A*, we describe it as *A*, *C*, *E*, *B*, *D*, *A*. The total cost of this tour is $741, which is a little better than the nearest-neighbor tour (see Example 6.4) but not as good as the repetitive nearest-neighbor tour (see Example 6.5).



ALGORITHM 4: THE CHEAPEST LINK ALGORITHM

- **Step 1.** Pick the *cheapest link* (i.e., edge with smallest weight) available. (In case of a tie, pick one at random.) Mark it (say in red).
- **Step 2.** Pick the next cheapest link available and mark it.

ALGORITHM 4: THE CHEAPEST LINK ALGORITHM

- **Step 3, 4, …, *N* – 1.** Continue picking and marking the cheapest unmarked link available that does not

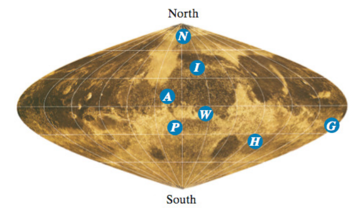
(a) close a circuit or

(b) (b) create three edges coming out of a single vertex

- **Step *N*.** Connect the last two vertices to close the red circuit. This circuit gives us the cheapest-link tour.

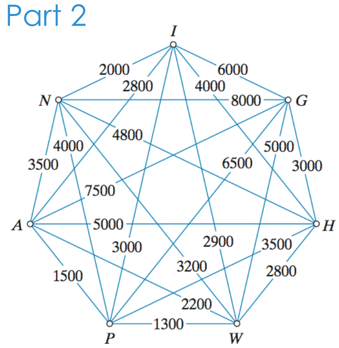
Example 6.7 Roving the Red Planet: Part 2

The figure shows seven sites on Mars identified as particularly interesting sites for geological exploration. Our job is to find an optimal tour for a rover that will land at *A*, visit all the sites to collect rock samples, and at the end return to *A*.



Example 6.7 Roving the Red Planet: Part 2

The distances between sites (in miles) are given in the graph shown.



Example 6.7 Roving the Red Planet: Part 2

Let's look at some of the approaches one might use to tackle this problem.

Brute force: The brute-force algorithm would require us to check and compute 6! = 720 different tours. We will pass on that idea for now.

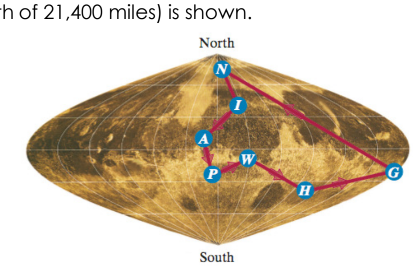
Cheapest link: The cheapest-link algorithm is a reasonable algorithm to use – not trivial but not too hard either. A summary of the steps is shown in the table on the next slide.

TABLE 6-7

Step	Cheapest edge available	Weight	Add to circuit?
1	*PW*	1300	Yes
2	*AP*	1500	Yes
3	*IN*	2000	Yes
4	*AW*	2200	No
5	*HW* } tie	2800	Yes
6	*AI*	2800	Yes
7	*IW*	2900	No &
8	*IP* } tie	3000	No &
9	*GH*	3000	Yes
Last	*GN* only way to close circuit	8000	Yes

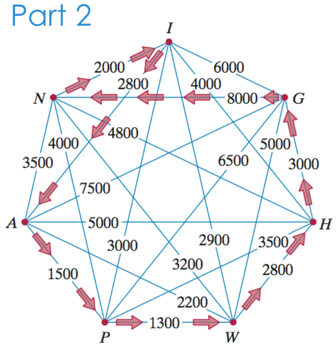
Example 6.7 Roving the Red Planet: Part 2

The cheapest-link tour (*A*, *P*, *W*, *H*, *G*, *N*, *I*, *A* with a total length of 21,400 miles) is shown.



Example 6.7 Roving the Red Planet: Part 2

Here is the cheapest-link tour again (*A*, *P*, *W*, *H*, *G*, *N*, *I*, *A* with a total length of 21,400 miles) is shown.

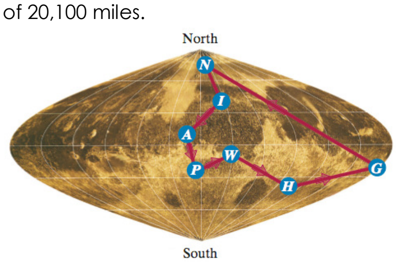


Example 6.7 Roving the Red Planet: Part 2

Nearest neighbor: The nearest-neighbor algorithm is the simplest of all the algorithms we learned. Starting from *A* we go to *P*, then to *W*, then to *H*, then to *G*, then to *I*, then to *N*, and finally back to *A*. The nearest-neighbor tour (*A*, *P*, *W*, *H*, *G*, *I*, *N*, *A* with a total length of 20,100 miles) is shown on the next two slides. (We know that we can repeat this method using different starting points, but we won't bother with that at this time.)

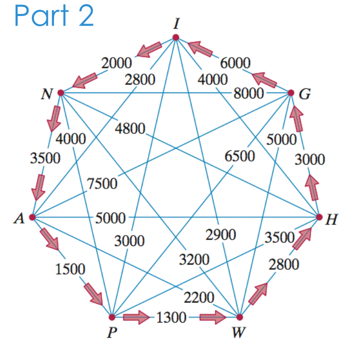
Example 6.7 Roving the Red Planet: Part 2

Nearest neighbor: *A*, *P*, *W*, *H*, *G*, *I*, *N*, *A* with a total length of 20,100 miles.



Example 6.7 Roving the Red Planet: Part 2

Nearest neighbor: *A*, *P*, *W*, *H*, *G*, *I*, *N*, *A* with a total length of 20,100 miles.



Surprise One

The first surprise in Example 6.7 is that the nearest-neighbor algorithm gave us a better tour than the cheapest-link algorithm. Sometimes the cheapest-link algorithm produces a better tour than the nearest-neighbor algorithm, but just as often, it's the other way around. The two algorithms are different but of equal standing – neither one can be said to be superior to the other one in terms of the quality of the tours it produces.

Surprise Two

The second surprise is that the nearest-neighbor tour *A*, *P*, *W*, *H*, *G*, *I*, *N*, *A* turns out to be an optimal tour. (This can be verified using a computer and the brute-force algorithm.) Essentially, this means that in this particular example, the simplest of all methods happens to produce the optimal answer–a nice turn of events. Too bad we can't count on this happening on a more consistent basis!