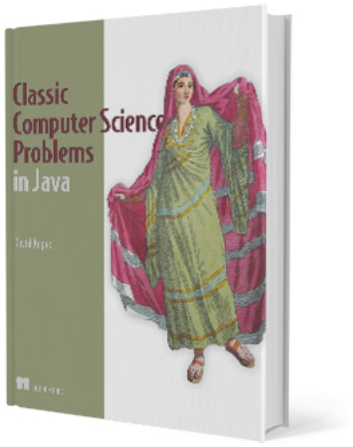
CODING

How to solve the classic Traveling Salesman Problem in Java

Sharpen your Java coding skills by exploring a well-known established computer science problem.

by David Kopec

July 23, 2021



[This article is excerpted from *Classic Computer Science Problems in Java*, Chapter 9, and is published with the kind permission of Manning Publications. —*Ed.*]

The Traveling Salesman Problem (TSP) is one of the most classic and talked-about problems in all of computing:

> A salesman must visit all the cities on a map exactly once, returning to the start city at the end of the journey. There is a direct connection from every city to every other city, and the salesman may visit the cities in any order. What is the shortest path for the salesman?

The problem can be thought of as a *graph problem*, with the cities being the vertices and the connections between them being the edges. Your first instinct might be to use a minimum spanning tree algorithm. Unfortunately, the solution to the Traveling Salesman Problem is not so simple. The minimum spanning tree is the way to connect all the cities with the least amount of road, but it does not provide the *shortest* path for visiting all of them exactly once.

Although the problem, as posed, appears simple, there is no algorithm that can solve it quickly for an arbitrary number of cities. What do I mean by “quickly”? I mean that the problem is what is known as *NP hard*. An NP-hard (nondeterministic polynomial hard) problem is a problem for which no *polynomial time algorithm* is known. (The time it takes is a polynomial function of the size of the input.)

As the number of cities that the salesman needs to visit increases, the difficulty of solving the problem grows exceptionally quickly. It is much harder to solve the problem for 20 cities than 10. It is impossible (to the best of current knowledge), in a reasonable amount of time, to solve the problem perfectly (optimally) for millions of cities.

The naive approach

The naive approach to the problem is simply to try every possible combination of cities. This approach to the TSP is $O(n!)$. Why this is the case is discussed in the “Taking it to the next level” section, but please don’t jump ahead, because the implementation of a naive solution to the problem will make its complexity obvious. Also, attempting the naive approach will illustrate the difficulty of the problem and this approach’s unsuitability for brute-force attempts at larger scales.

Our sample data. In our version of the TSP, the salesman is interested in visiting five of the major cities in Vermont. We will not specify a starting (and therefore ending) city. **Figure 1** illustrates the five cities and the driving distances between them. Note that there is a distance listed for the route between every pair of cities.

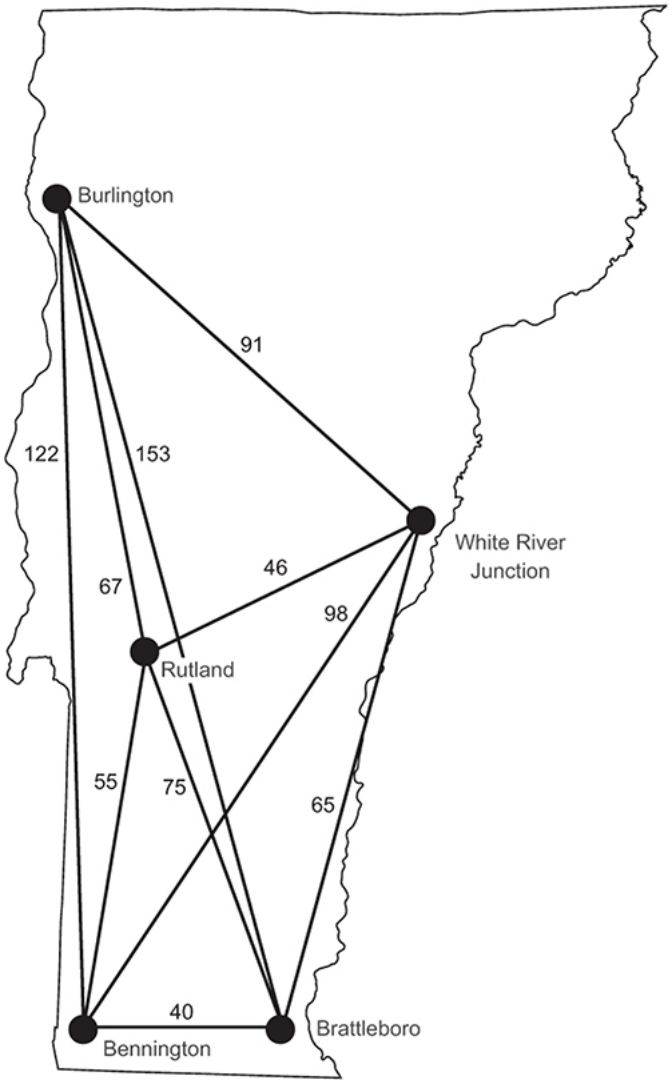


Figure 1. Five cities in Vermont and the driving distances between them

Perhaps you have seen driving distances in table form before. In a driving-distance table, one can easily look up the distance between any two cities. **Table 1** lists the driving distances for the five cities in the problem.

	Rutland	Burlington	White River Junction	Bennington	Brattleboro
Rutland	0	67	46	55	75
Burlington	67	0	91	122	153
White River Junction	46	91	0	98	65
Bennington	55	122	98	0	40
Brattleboro	75	153	65	40	0

Table 1. Driving distances between cities in Vermont

We will need to codify both the cities and the distances between them for our problem. To make the distances between cities easy to look up, we will use a map of maps, with the outer set of keys representing the first of a pair and the inner set of keys representing the second. This will be the type `Map<String, Map<String, Integer>>`, and it will allow lookups like `vtDistances.get("Rutland").get("Burlington")`, which should return 67.

We will use the `vtDistances` map when we solve the problem for Vermont, but first, let's do some setup. Our class holds the map and has a utility method we will use later for doing a swap of the items at two locations within an array.

Let's start off with the top part of the listing for TSP.java, as follows (See my GitHub repository for all the code):

```
package chapter9;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Map;

public class TSP {
    private final Map<String, Map<String, Integer>

    public TSP(Map<String, Map<String, Integer>> d
        this.distances = distances;
    }

    public static <T> void swap(T[] array, int fir
        T temp = array[first];
        array[first] = array[second];
        array[second] = temp;
    }
```

Finding all permutations. The naive approach we are using to solve the TSP requires generating every possible permutation of the cities. There are many permutation-generation algorithms; they are simple enough to ideate that you could certainly come up with one on your own.

One common approach is *backtracking*, such as is used to solve a constraint-satisfaction problem. In constraint-satisfaction problem-solving, backtracking is used after a partial solution is found that does not satisfy the problem's constraints. In such a case, you revert to an earlier state and continue the search along a different path than that which led to the incorrect partial solution.

To find all of the permutations of the items in an array (eventually, the cities), we will also use backtracking. After we make a swap between elements and go down a path of further permutations, we will backtrack

to the state before the swap was made so we can make a different swap and go down a different path.

Here is more of TSP.java

```
    private static <T> void allPermutationsHelper(T[]
            // Base case - we found a new permutation,
            if (n <= 0) {
                permutations.add(permutation);
                return;
            }
            // Recursive case - find more permutations
            T[] tempPermutation = Arrays.copyOf(permut
            for (int i = 0; i < n; i++) {
                swap(tempPermutation, i, n - 1); // mo
                // move everything else around, holdir
                allPermutationsHelper(tempPermutation,
                swap(tempPermutation, i, n - 1); // ba
            }
        }
```

This recursive function is labeled a “helper,” because it will be called by another function that takes fewer arguments. The parameters of `allPermutationsHelper()` are the starting permutation we are working with, the permutations generated so far, and the number of remaining items to swap around.

A common pattern for recursive functions that need to keep multiple items of state across calls is to have a separate outward-facing function with fewer parameters that is easier to use. `allPermutations()` is that simpler function.

More of TSP.java

```
    private static <T> List<T[]> permutations(T[] orig
            List<T[]> permutations = new ArrayList<>()
            allPermutationsHelper(original, permutatio
            return permutations;
        }
```

`allPermutations()` takes just a single argument: the array for which the permutations should be generated. It calls `allPermutationsHelper()` to find those permutations. This saves the user of `allPermutations()` from having to provide the parameters’ permutations and *n* to `allPermutationsHelper()`.

The backtracking approach to finding all the permutations presented here is fairly efficient. Finding each permutation requires just two swaps within the array. However, it is possible to find all the permutations of an array with just one swap per permutation. One efficient algorithm that accomplishes that task is Heap’s algorithm. (This is not to be confused with the heap data structure. Heap, in this case, is the name of the inventor of the algorithm, B.R. Heap, who described it in 1963. Learn more about that in Robert Sedgewick’s paper “Permutation Generation Methods.”)

This difference in efficiency may be important for very large data sets—which is not what we are dealing with here, of course.

Brute-force search. We can now generate all the permutations of the city list, but this is not quite the same as a TSP path. Recall that in the TSP, the salesman must return, at the end, to the same city that he started in. We can easily add the distance from the last city the salesman visited to the first city visited when we calculate which path is the shortest, and we will do that shortly.

We are now ready to try testing the paths we have permuted. A brute-force search approach painstakingly looks at every path in a list of paths and uses the distance between the two cities in the lookup table (distances) to calculate each path's total distance. It prints both the shortest path and that path's total distance.

More from TSP.java

```
public int pathDistance(String[] path) {
        String last = path[0];
        int distance = 0;
        for (String next : Arrays.copyOfRange(path
            distance += distances.get(last).get(ne
            // distance to get back from last city
            last = next;
        }
        return distance;
    }

    public String[] findShortestPath() {
        String[] cities = distances.keySet().toArr
        List<String[]> paths = permutations(cities
        String[] shortestPath = null;
        int minDistance = Integer.MAX_VALUE; // ar
        for (String[] path : paths) {
            int distance = pathDistance(path);
            // distance from last to first must be
            distance += distances.get(path[path.le
            if (distance < minDistance) {
                minDistance = distance;
                shortestPath = path;
            }
        }
        // add first city on to end and return
        shortestPath = Arrays.copyOf(shortestPath,
        shortestPath[shortestPath.length - 1] = sh
        return shortestPath;
    }

    public static void main(String[] args) {
        Map<String, Map<String, Integer>> vtDistar
                "Rutland", Map.of(
                        "Burlington", 67,
                        "White River Junction", 46
                        "Bennington", 55,
                        "Brattleboro", 75),
                "Burlington", Map.of(
                        "Rutland", 67,
                        "White River Junction", 91
                        "Bennington", 122,
                        "Brattleboro", 153),
                "White River Junction", Map.of(
                        "Rutland", 46,
                        "Burlington", 91,
                        "Bennington", 98,
                        "Brattleboro", 65),
                "Bennington", Map.of(
                        "Rutland", 55,
                        "Burlington", 122,
                        "White River Junction", 98
```

```
                        "Brattleboro", 40),
                "Brattleboro", Map.of(
                        "Rutland", 75,
                        "Burlington", 153,
                        "White River Junction", 65
                        "Bennington", 40));
        TSP tsp = new TSP(vtDistances);
        String[] shortestPath = tsp.findShortestPa
        int distance = tsp.pathDistance(shortestPa
        System.out.println("The shortest path is "
                distance + " miles.");
    }
}
```

We finally can brute-force the cities of Vermont, finding the shortest path to reach all five. The output should look something like the following, and the best path is illustrated in **Figure 2**.

The shortest path is [White River Junction, Burlington, Rutland, Bennington, Brattleboro, White River Junction] 318 miles.

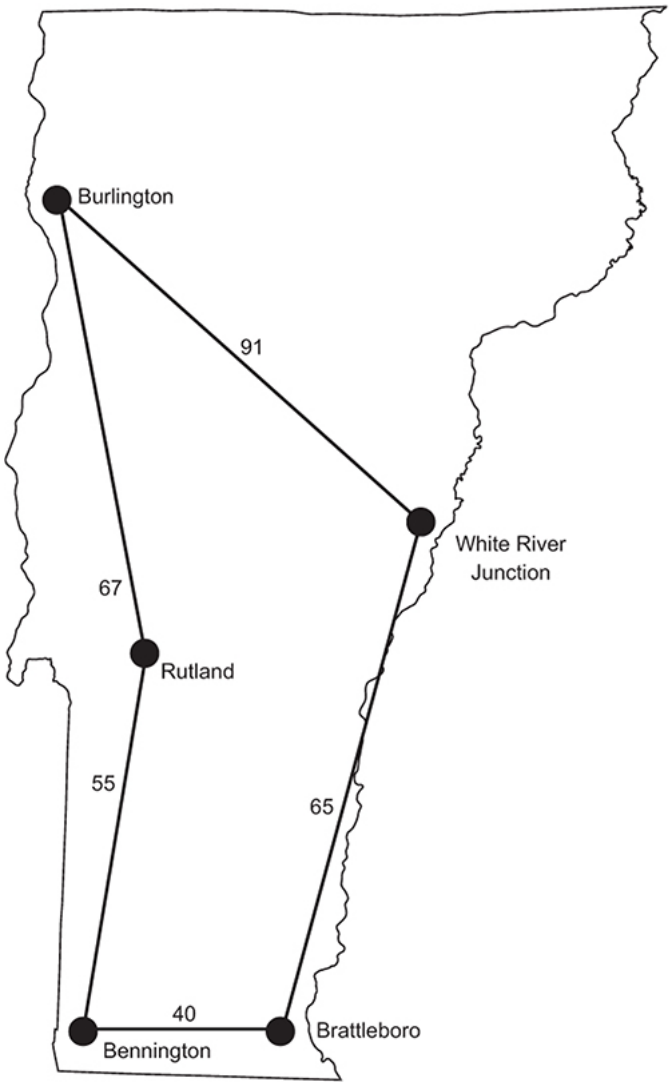


Figure 2. The shortest path for the salesman to visit all five cities in Vermont is illustrated.

Taking it to the next level

There is no easy answer to the TSP. Our naive approach quickly becomes infeasible. The number of permutations generated is *n* factorial (*n*!), where *n* is the number of cities in the problem. If we were

to include just one more city (six instead of five), the number of evaluated paths would grow by a factor of six. Then it would be seven times harder to solve the problem for just one more city after that. This is not a scalable approach!

In the real world, the naive approach to the TSP is seldom used. Most algorithms for instances of the problem with many cities are approximations. They try to solve the problem for a near-optimal solution. The near-optimal solution may be within a small known band of the perfect solution. (For example, perhaps they will be no more than 5% less efficient.)

The TSP is an everyday occurrence for shipping and distribution companies like UPS and FedEx. Package delivery companies want their drivers to travel the shortest routes possible. Not only does this make the driver's job more pleasant, but it also saves fuel and maintenance costs. We all travel for work or for pleasure, and finding optimal routes when visiting many destinations can save resources.

But the TSP is not just for routing travel; it comes up in almost any routing scenario that requires singular visits to nodes. Although a minimum spanning tree may minimize the amount of wire needed to connect a neighborhood, it does not tell us the optimal amount of wire if every house must be forward-connected to just one other house as part of a giant circuit that returns to its origination. The TSP does.

Permutation and combination generation techniques, like the ones used in the naive approach to the TSP, are useful for testing all sorts of brute-force algorithms. For instance, if you were trying to crack a short password and you knew its length, you could generate every possible permutation of the characters that could potentially be in the password. Practitioners of such large-scale permutation-generation tasks would be wise to use an especially efficient permutation-generation algorithm such as Heap's algorithm.

Dig deeper

- Permutations by interchanges, by B.R. Heap, *The Computer Journal*, November 1963
- Optimizing grocery delivery routes using Oracle APEX, Oracle Autonomous Database, and Google Maps APIs
- NP-Completeness

David Kopec

David Kopec (@davekopec) is assistant professor of Computer Science and Innovation at Champlain College in Burlington, Vermont. He is the author of *Classic Computer Science Problems in Java* (Manning, 2020) and *Classic Computer Science Problems in Python* (Manning, 2019), among others. He is also a software developer and podcaster.

Share this Page