Assignment 3: Traveling Salesperson

COSC 3020: Algorithms and Data Structures

Lars Kotthoff
larsko@uwyo.edu

07 November 2018

Instructions

Solve the following tasks. You may work in teams of up to two people. For the theoretic part, submit a PDF with your solution, for the practical part, submit the JavaScript file(s) to WyoCourses. The name and student ID of *all* partners must be clearly visible on each page of your PDF and in each source code file. If you do not submit a PDF file and *separate* source file(s), you may lose points. If your code does not work with `node.js` on Linux, you may lose points. Only one partner needs to submit. You have until Friday, 30 November 2018, 23:59h.

You may *not* use external libraries in your code unless explicitly stated.

This assignment is about the Traveling Salesperson Problem I mentioned in the lecture on NP-hard problems. We consider the version of the Traveling Salesperson Problem that finds the shortest tour to visit n cities, starting at a city and ending at the nth city; it *does not* go back to the start. The start city may be any of the cities.

1 Dynamic Programming (15 points)

The Held-Karp algorithm for solving the Traveling Salesperson Problem is a recursive algorithm that considers every subset of cities and finds shortest tours within them. It takes advantage of the fact that every subroute of a route of minimum length is of minimum length itself. The main idea is that to solve the problem of finding the shortest route for n cities, we first solve the problem of finding the shortest route for $n-1$ cities, and then find the shortest route from the $n-1$st city to the nth city. The pseudocode for the algorithm is as follows:

```
// cities is the set of cities not visited so far, including
    start
heldKarp(cities, start)
  if |cities| == 2
    return length of tour that starts at start, goes directly
       to other city in cities
  else
    return the minimum of
      for each city in cities, unless the city is start
        // reduce the set of cities that are unvisited by one
            (the old start), set the new start, add on the
            distance from old start to new start
        heldKarp(cities - start, c) + distance from start to c
```

Implement a dynamic programming version (which could use memoization) of the Held-Karp algorithm. Investigate your implementation's empirical time complexity, i.e. how the runtime increases as the input size increases. *Measure* this time by running the code instead of reasoning from the asymptotic complexity (this is the empirical part). Create inputs of different sizes and plot how the runtime scales (input size on the x axis, time on the y axis). Your largest input should have a runtime of *at least* an hour. I recommend running this code in the cloud so you can explore much larger inputs.

The choice of data structures for representing the graph, distances, etc. is up to you. The only requirement is that your algorithm, given a set of cities and their distances, outputs the length of the optimal tour. Submit your implementation along with the code you used for your experiments and a report detailing your experiments and your runtime plot.

What is the worst-case asymptotic time complexity of your implementation? What is the worst-case asymptotic memory complexity?

2 Stochastic Local Search (15 points)

The 2-opt algorithm for solving the Traveling Salesperson Problem is a randomized local search algorithm that, at each iteration, reverses part of the route. It starts with a random route (this is the randomized part), and changes part of the route in each step (this is the local search part, sprinkled with more randomness). The pseudocode for one iteration is as follows:

```
2optSwap(route, i, k)
  cities 1 to i-1 stay in the order they are
  cities i to k are reversed
  cities k + 1 to n stay in the order they are
```

For example, if I call the above function with route A–B–C–D–E–F, $i = 2$, $k = 4$, the resulting route is A–B–E–D–C–F.

The algorithm starts with a random route; if the new route at the end of an iteration decreases the total length, it is retained as the current incumbent. The incumbent after the final iteration is returned as the solution.

Implement the 2-opt algorithm, which repeatedly runs the above steps. Your implementation needs to fix two design parameters that I have left open. First, you need to design a stopping criterion – when would it make sense to stop and return the shortest route found so far rather than try another iteration? Second, design a way to choose i and k – note that they need to be different between different iterations, as one iteration would simply undo what the previous one did otherwise.

Investigate the empirical time complexity of your implementation, i.e. how the runtime increases as the input size increases. Create inputs of different sizes and plot how the runtime scales (input size on the x axis, time on the y axis). Your largest input should have a runtime of *at least* an hour.

Again the choice of data structures is up to you, but you probably want to use the same inputs as in the previous question. Submit your implementation along with the code you used for the experiments and a report detailing your experiments and your runtime plot.

What is the worst-case asymptotic time complexity of your implementation? What is the worst-case asymptotic memory complexity?

3 Comparison (5 points)

How does the empirical time complexity of your the Held-Karp and the 2-opt algorithms compare? Note that you need to run the two codes on the same graphs and the same machine to be comparable. How do the achieved solution qualities, i.e. the lengths of the shortest tours, compare? A good way to show this is to have a plot that show the values for both at the same time, either as absolute values or relative to each other.