Dynamic Programming Solution for TSP

CS 535 Design and Analysis of Algorithms
Fall Semester, 2018

Dynamic programming does not always yield polynomial-time algorithms (a fact that the text does not discuss), but even when it gives exponential-time algorithms, it can provide a useful framework for algorithm design. In this section we’ll look at such an example, the Travelling Salesman Problem, or TSP for short.


In the TSP, we are given \( n \) cities and a distance matrix \( C \) in which \( C_{ij} \) is the distance from city \( i \) to city \( j \). These distances do not have to correspond to physical distances; they are costs. Because they are costs, they do not need to satisfy constraints such as symmetry (the distance from \( i \) to \( j \) need not be the same as the distance from \( j \) to \( i \)) or the triangle inequality (it is not necessarily shorter to go from \( i \) to \( j \) directly—it mean be cheaper to go from \( i \) to \( k \) to \( j \)). We need to route a salesman from his home city, arbitrarily called city 1, through all other \( n - 1 \) cities and then back home to city 1. Such a routing is called a tour. The TSP is to find the cheapest tour.

There are \((n - 1)!\) possible tours (why?), so a brute-force examination of the cost of each tour would take time \( (n - 1) \times (n - 1)! \) because it would take \( n - 1 \) additions to compute the cost of each tour (this could be improved by choosing successive tours that differ only by the interchange of two cities). With a dynamic programming approach we can improve this time bound to \( O(n^2 \times 2^n) \).

We define

\[
T(i; j_1, j_2, \ldots, j_k) = \begin{cases} 
\text{cost of the optimal tour from city } i \text{ to city 1 that goes through each of the intermediate cities } j_1, j_2, \ldots, j_k \text{ exactly once, in any order, and through no other cities.} 
\end{cases}
\]

The optimal substructure property tells us that

\[
T(i; j_1, j_2, \ldots, j_k) = \min_{1 \leq m \leq k} \{C_{ij_m} + T(j_m; j_1, j_2, \ldots, j_{m-1}, j_{m+1}, \ldots, j_k)\}.
\]

Furthermore,

\[
T(i; \emptyset) = C_{i1}.
\]

The value we want is

\[
T(1; 2, 3, \ldots, n).
\]

Without memoization, direct evaluation of \( T(i; j_1, j_2, \ldots, j_k) \) takes time \( \Theta(k) \) time plus the time for the \( k \) recursive calls; if \( t(k) \) is the order of the time needed for \( k \) intermediate cities,

\[
t(k) = k + k \times t(k - 1)
\]

and \( t(0) \) is constant. The overall time is then \( t(n - 1) > (n - 1)! \).

With memoization, there are \( n - 1 \) choices for city \( i \) and \( \binom{n-2}{k} \) sets of \( k \) intermediate cites chosen from among all cities except cities 1 and \( i \). The total number of memos [including the memo for \( T(1; 2, 3, \ldots, n) \)] is thus

\[
1 + \sum_{k=1}^{n-2} (n - 1) \binom{n-2}{k} = 1 + (n - 1) \sum_{k=1}^{n-2} \binom{n-2}{k} = 1 + (n - 1)(2^{n-2} - 1).
\]
Evaluating a memo once all the needed memos are available is $O(n)$, so the overall cost of the memoized algorithm is $O(n^2 2^n)$.