Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

\[\text{Cost}(2, 2) = \min\{\text{c}(2, 4) + \text{Cost}(4, 8) + \text{Cost}(8, 9), \text{c}(2, 6) + \text{Cost}(6, 8) + \text{Cost}(8, 9)\} = 8\]

\[\text{Cost}(2, 3) = \{\text{c}(3, 4) + \text{Cost}(4, 8) + \text{Cost}(8, 9), \text{c}(3, 5) + \text{Cost}(5, 8) + \text{Cost}(8, 9), \text{c}(3, 6) + \text{Cost}(6, 8) + \text{Cost}(8, 9)\} = 10\]

Step-3: Cost (K-4, j)

\[\text{Cost}(1, 1) = \{\text{c}(1, 2) + \text{Cost}(2, 6) + \text{Cost}(6, 8) + \text{Cost}(8, 9), \text{c}(1, 3) + \text{Cost}(3, 5) + \text{Cost}(5, 8) + \text{Cost}(8, 9)\} = 12\]

\[\text{c}(1, 3) + \text{Cost}(3, 6) + \text{Cost}(6, 8) + \text{Cost}(8, 9) = 13\]

Hence, the path having the minimum cost is 1 → 3 → 5 → 8 → 9.

DAA - Travelling Salesman Problem

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are \((n - 1)!\) number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph \(G = (V, E)\), where \(V\) is a set of cities and \(E\) is a set of weighted edges. An edge \(e(u, v)\) represents that vertices \(u\) and \(v\) are connected. Distance between vertex \(u\) and \(v\) is \(d(u, v)\), which should be non-negative.

Suppose we have started at city 1 and after visiting some cities now we are in city \(j\). Hence, this is a partial tour. We certainly need to know \(j\), since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don’t repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities \(S \subseteq \{1, 2, 3, \ldots, n\}\) that includes 1, and \(j \in S\), let \(C(S, j)\) be the length of the shortest path visiting each node in \(S\) exactly once, starting at 1 and ending at \(j\).

When \(|S| > 1\), we define \(C(S, 1) = \infty\) since the path cannot start and end at 1.

Now, let express \(C(S, j)\) in terms of smaller sub-problems. We need to start at 1 and end at \(j\). We should select the next city in such a way that

\[C(S, j) = \min\{C(S - \{j\}, i) + d(i, j) \text{ where } i \in S \text{ and } i \neq j\}

Analysis

There are at the most \(2^n \cdot n\) sub-problems and each one takes linear time to solve. Therefore, the total running time is \(O(2^n \cdot n^2)\).

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.
From the above graph, the following table is prepared.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>13</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

**S = Φ**

\[
\begin{align*}
\text{Cost}(2, Φ, 1) &= d(2, 1) = 5\text{Cost}(2, Φ, 1) = d(2, 1) = 5 \\
\text{Cost}(3, Φ, 1) &= d(3, 1) = 6\text{Cost}(3, Φ, 1) = d(3, 1) = 6 \\
\text{Cost}(4, Φ, 1) &= d(4, 1) = 8\text{Cost}(4, Φ, 1) = d(4, 1) = 8
\end{align*}
\]

**S = 1**

\[
\begin{align*}
\text{Cost}(i, s) &= \min\{\text{Cost}(j, s - (j)) + d[i, j]\} \\
\text{Cost}(i, s) &= \min\{\text{Cost}(j, s) - (j)) + d[i, j]\}
\end{align*}
\]

\[
\begin{align*}
\text{Cost}(2, \{3\}, 1) &= d[2, 3] + \text{Cost}(3, Φ, 1) = 9 + 6 = 15\text{Cost}(2, \{3\}, 1) = d[2, 3] \\
&+ \text{cost}(3, Φ, 1) = 9 + 6 = 15 \\
\text{Cost}(2, \{4\}, 1) &= d[2, 4] + \text{Cost}(4, Φ, 1) = 10 + 8 = 18\text{Cost}(2, \{4\}, 1) = d[2, 4] \\
&+ \text{cost}(4, Φ, 1) = 10 + 8 = 18 \\
\text{Cost}(3, \{2\}, 1) &= d[3, 2] + \text{Cost}(2, Φ, 1) = 13 + 5 = 18\text{Cost}(3, \{2\}, 1) = d[3, 2] \\
&+ \text{cost}(2, Φ, 1) = 13 + 5 = 18 \\
\text{Cost}(3, \{4\}, 1) &= d[3, 4] + \text{Cost}(4, Φ, 1) = 12 + 8 = 20\text{Cost}(3, \{4\}, 1) = d[3, 4] \\
&+ \text{cost}(4, Φ, 1) = 12 + 8 = 20 \\
\text{Cost}(4, \{3\}, 1) &= d[4, 3] + \text{Cost}(3, Φ, 1) = 9 + 6 = 15\text{Cost}(4, \{3\}, 1) = d[4, 3] \\
&+ \text{cost}(3, Φ, 1) = 9 + 6 = 15 \\
\text{Cost}(4, \{2\}, 1) &= d[4, 2] + \text{Cost}(2, Φ, 1) = 8 + 5 = 13\text{Cost}(4, \{2\}, 1) = d[4, 2] \\
&+ \text{cost}(2, Φ, 1) = 8 + 5 = 13
\end{align*}
\]

**S = 2**

\[
\begin{align*}
\text{Cost}(2, \{3, 4\}, 1) &=
\begin{cases}
  d[2, 3] + \text{Cost}(3, \{4\}, 1) = 9 + 20 = 29 \\
  d[2, 4] + \text{Cost}(4, \{3\}, 1) = 10 + 15 = 25 = 25\text{Cost}(2, \{3, 4\}, 1)
\end{cases}
\end{align*}
\]
The minimum cost path is 35.

Start from \( \{1, 2, 3, 4\}, 1 \), we get the minimum value for \( d[1, 2] \). When \( s = 3 \), select the path from 1 to 2 (cost is 10) then go backwards. When \( s = 2 \), we get the minimum value for \( d[4, 2] \). Select the path from 2 to 4 (cost is 10) then go backwards.

When \( s = 1 \), we get the minimum value for \( d[4, 3] \). Selecting path 4 to 3 (cost is 9), then we shall go to then go to \( s = \Phi \) step. We get the minimum value for \( d[3, 1] \) (cost is 6).

**DAA - Optimal Cost Binary Search Trees**

A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes. The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.

When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node.

Search time of an element in a BST is \( O(n) \), whereas in a Balanced-BST search time is \( O(\log n) \). Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

Here, the Optimal Binary Search Tree Algorithm is presented. First, we build a BST from a set of provided \( n \) number of distinct keys \( < k_1, k_2, k_3, ..., k_n > \). Here we assume, the probability of accessing a key \( K_i \) is \( p_i \). Some dummy keys \( (d_0, d_1, d_2, ..., d_n) \) are added as some searches may be performed for the values which are not present in the Key set \( K \). We assume, for each dummy key \( d_i \) probability of access is \( q_i \).