A*를 이용한 최적여행경로제공 웹사이트 구현

동국대학교 관광대학 관광레저개발학과*, 자연과학대학 컴퓨터학과**
서 태 양*·임 재 결**

Implementation of a Web Site Providing Optimal Touring Routes using A*

Tai-Yang Seo*·Jae-Geol Yim**

*Dean. of College of Tourism at Dongguk University
**Department of Computer Science, College of Natural Science
Dongguk University, Kyongju 780-714, Korea

= 초 록 =

사용자가 국도로 여행한다는 가정 하에, 사용자에게 '여행 최적 경로'를 찾아주는 www 웹페이지를 구축하였다. 본 웹페이지는 사용자에게 한국 지도를 보여주고, 출발지와 종착지를 비롯한 경유지를 선택하도록 한 다음, 이를 경유하는 최단거리 경로를 찾아서 지도상에 보여준다. 최단 거리 경로를 찾는 문제는 효율적인 해가 없는 어려운 문제로 유명하다. 본 논문은 최단거리를 빠르게 찾기 위하여 히스틱 알고리즘을 사용하였다. 기존의 최단거리를 찾는 히스틱 알고리즘은 출발지와 종착지가 동일한 경우에만 해를 찾는다. 이에 반하여, 본 논문에서 사용한 알고리즘은 출발지와 종착지가 서로 다르더라도 최적해를 구하는 특징이 있다.

핵심용어: WWW(월드와이드웹), 히스틱 탐색, 최단경로, 최적경로, JAVA

* 동국대학교 관광대학 학장과 대한관광경영학회 회장으로 재직 중이다. 관광상을 개발 및 문화 관광, 관광지 관리에 관심을 갖고 있다. [sty@dongguk.ac.kr]
** 미국 인리노이주립대학에서 인공지능으로 박사학위를 취득하였으며 현재 동국대학교 창업보육센터 소장 겸 컴퓨터학과 교수로 재직 중임. WWW, 컴퓨터망, 인공지능, 및 일터미디어 이론 및 응용에 관심을 갖고 있다. [yim@dongguk.ac.kr]
I. INTRODUCTION

It is predicted that tourism will be one of the most promising industries in the 21st Century (Hunt, 1987). Most countries in the world have spent their best efforts to activate tourism. The greatest portion of their efforts and money has been invested in developing tourism facilities, and in the improvement of transportation installations. These kinds of efforts sometimes result in conflicts against natural resources.

Tourism development has recently been interpreted to enlarge the value of tourism resources. Therefore, developing information systems, nature preservation, and tourism information provisions are a form of tourism development (Seo, 1996). Nowadays, the internet is so popular that the younger generations all over the world enjoy surfing on it, and information provisions on the internet have become a promising business. The authors of this paper also participated in developing and administrating two WWW sites; one providing tourism information on the Kyung-Buk province in Korea (http://www.kyongbuktour.or.kr) and the other on the Bul-kuk temple (http://www.bulguksa.or.kr).

Providing raw data on the WWW can be done by using HTML (http://www.w3.org/pub/WWW/MarkUp/MarkUp.html), and is a relatively easy task, although providing useful information is not. This paper introduces a method to manipulate raw data, and come up with useful information, particularly an optimal route to provide the user on the WWW.

When a user opens our homepage, a map of the southern half of Korea appears as well as input-out boxes, and a list of all the names of cities. The user chooses a start city, destination, and all the cities he wants to visit. The start city may be different from the destination. The homepage then returns a sequence of city names, and draws the route corresponding to the sequence on the map. The sequence of cities is optimal in the sense of distance, i.e., travel will be minimal if he visits cities in the order of the sequence. The problem of finding such a sequence can be modeled as a TSP problem (Traveling Sales Person). As a well-known NP-complete problem,
TSP has been studied by many researchers. A few heuristic algorithms for TSP are described in the “Fundamentals of Computer Algorithms” (Horowitz and Sahni, 1984). The worst case complexities of these algorithms are also exponential. However, these are fast enough for practical problems of no more than 50 cities (Har, Nilsson, and Raphael, 1968). No matter how many cities are on the map, as long as there are less than 50 cities the user wants to visit, then the algorithm works fine.

Making use of these algorithms, our home page finds an optimal course. However, our method is different from the existing algorithms in that the starting city and the destination may not be the same. All algorithms for TSP require the shortest paths between all possible pairs of cities. With our homepage, the shortest paths between all possible pairs (170 × 170 = 28,900 pairs) of over 170 cities are on the map. To get these shortest paths we have implemented a modified version of Warshall’s algorithm as a computer program in JAVA language. Warshall’s algorithm finds the cost of the shortest path; while we have modified Warshall’s algorithm so that the new version finds both the cost and the shortest path. The input data for our version of Warshall’s algorithm is the adjacency matrix for the 170 cities. The adjacency matrix is a 170 × 170 matrix. Rows and columns are corresponding to cities. The i-th row and the i-th column correspond to the same city. The typical entry aij of the adjacency matrix is defined as 0 if i = j, infinite if there is no direct route from city i to city j, and the length of the road if there is a direct route from city i to city j. We call it a directed route from city i to city j if there is no other city on the route. For better understanding, an example of an adjacency matrix is given in Table 1. The adjacency matrix is for 26 virtual cities, A, B, C, Z. The first row of the matrix represents that there is no direct route from A to B, D, X, Y, Z. There are roads of length 3 and 5 from A to C and E, respectively. Similarly there is a road of length 14 between B and X.

Our system has been implemented in JAVA language. This paper describes how it has been done.
<Table 1> An example adjacency matrix.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>...</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>∞</td>
<td>3</td>
<td>∞</td>
<td>5</td>
<td></td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
<td>14</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td></td>
<td>∞</td>
<td>4</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>8</td>
<td>∞</td>
<td></td>
<td>∞</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>8</td>
<td>0</td>
<td></td>
<td>∞</td>
<td>∞</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>∞</td>
<td>14</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
<td>0</td>
<td>4</td>
<td>∞</td>
</tr>
<tr>
<td>Y</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td></td>
<td>4</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>Z</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>9</td>
<td>6</td>
<td></td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

II. THEORETICAL BACKGROUND

1. Warshall’s Algorithm

Given the adjacency matrix C, Warshall’s algorithm (Horowitz and Sahni, 1984) finds a matrix A such that A(i, j) is the length of a shortest path from city i to city j. Examining the shortest path from I to j, this path originates at city I, and goes through some intermediate cities terminating at city j. If k is an intermediate city on the shortest path, then the subpaths from I to k, and from k to j must be the shortest paths from I to k, and k to j respectively. Otherwise, the I to j path is not of minimum length. If k is the intermediate city with the highest index, then the I to k path is the shortest I to k path in G, going through no vertex with an index greater than k - 1. Similarly, the k to j path is the shortest k to j path in G, going through no vertex of an index greater than k-1. We may regard the construction of the shortest I to j path as first requiring a decision as to which the highest indexed intermediate city k is.

Once this decision has been made, we need to find the two shortest paths. One from I to k, and the other from k to j. Neither of these may go
through a vertex with an index greater than k-1. Let $A_k(I, j)$ be a matrix representing the length of the shortest path from I to j, going through no vertex of an index greater than k; when k=0, $A_0(I, j) = C(I, j)$. Warshall’s algorithm increases k one at a time, from 0 to n (the number of cities), and finally computes $A_n = A$. Warshall’s algorithm is shown in Table 2. By applying the algorithm to our adjacency matrix for 170 cities on the southern half of Korean peninsula, we obtain the lengths of all the shortest routes between all the pairs of cities.

<Table 2> Warshall’s Algorithm

```
Warshall(C, A, n)
// C is the adjacency matrix for 170 cities
// A(I,j) is the cost of a shortest path from city I to city j.
begin
    for I := 1 to n do
        for j := 1 to n do
            $A(I,j) := C(I,j)$
        for k := 1 to n do
            for I := 1 to n do
                for j := 1 to n do
                    $A(I,j) := \min(A(I,j), A(I,k)+A(k,j))$
end.
```

2. Algorithm $A^*$

Given a set of cities, the purpose of TSP is to find the optimal path to visit all the cities exactly once. The simplest way of solving this problem is to enumerate all the permutations of the given cities, compute the distances, then find the minimum. Simple methods are usually time-consuming. In this case, given n cities, the method has to enumerate n! permutations. Enumerating n! permutations by a computer is impossible, because of the shortage of memory or of CPU time.

This problem has been studied in Artificial Intelligence (AI) areas. One of the best methods is $A^*$ algorithm (Gelperin, 1977). It gradually expands
search space graph as it proceeds. The graph consists of a set of nodes. The first node is called the root node, and represents the starting city. To the root node, the algorithm adds one node (nodei) per adjacent city (cityi) to the starting city. Each of the added nodes, nodei, means that the traveler has moved to the corresponding city (cityi). Then, for each nodei, it estimates the cost of a TSP tour, in which the first move is the visiting cityi. We denote the value of the estimation hi. A function used for estimation is called a heuristic function, because there is no way to calculate the exact minimal cost. Now, it selects the node with minimum estimation value, and regarding it as a root node, continues adding new nodes to the selected node in the same manner as described above, until a node corresponding to the situation of all the cities have been visited is added. The skeleton of A* is shown in table 3 (Nilsson, 1980).

It is known that A* always finds a correct answer if the evaluation function is admissible, i.e. the function value is not greater than the real cost.

*Table 3* A skeleton of A* algorithm

Procedure A*

1. Create a search graph, G, consisting solely of the start node, s. Put s on a list called OPEN.
2. Create a list called CLOSED that is initially empty.
3. LOOP: if OPEN is empty, exit with failure.
4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node n.
5. If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G. (Pointers are established in step 7.)
6. Expand node n, generating the set, M, of its successors and install them as successors of n in G.
7. Establish a pointer to n from those members of M that were not already in G (i.e., not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member of M that was already on OPEN or CLOSED, decide whether or not to redirect its pointer to n. For each member of M already on CLOSED, decide for each of its descendants in G whether or not to redirect its pointer.
8. Reorder the list OPEN, according to heuristic merit.
9. Go LOOP.
A* algorithm is a general method for approaching problems of exponential complexity.

In order to apply A* to a problem, we need a heuristic function for the estimation at Step 8. If the heuristic function value is close to the real cost, then the A* using the heuristic function finds a solution quickly. On the other hand, if the heuristic function value is far different from the real cost, then the A* becomes a little better than a "blind search".

For a TSP problem, a heuristic function using the reduced cost matrix is well known. A row (column) is said to be reduced if, and only if it contains at least one zero entry. A matrix is reduced if, and only if every row and column is reduced. Since the tour we are looking for visits and leaves every city exactly once, a lower bound of the tour is the sum of minimum entries of all the rows. If t is chosen to be the minimum entry in row I, then subtracting t from every entry of row I will reduce row i. Doing this to every row will reduce all rows. After reducing all rows, we can reduce all the columns in the same manner. The total of the values, subtracted from all the rows and columns, is a better lower bound of the tour. For example, let suppose a user wants to visit all the cities, A, B, C, D, E exactly once, and the cost matrix for these five cities is as shown in Table 4. For every pair of cities u and v, the cost of the shortest path from u to v is shown. We find these minimum costs by applying Warshall’s algorithm on the adjacency matrix (Table 1).

We put at every diagonal entry, to avoid it from immediately coming back to the same city. For the cost matrix, shown in Table 4, we subtract 3, 4, 2, 2, 5 from rows 1, 2, 3, 4, 5 to reduce all rows. Then, we subtract 1 from the second column, and 3 from the fourth column to get the reduced matrix (Table 5). The sum of 3, 4, 2, 2, 5, 1, and 3 is 20, and implies that the minimum cost of a tour for this TSP problem is not less than 20. It is well known that using the total amount subtracted (20 in this example) as a heuristic function value at step 8 of A*, we can handle most practical TSP
problems.

<table>
<thead>
<tr>
<th>Table 4</th>
<th>An example cost matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5</th>
<th>The reduced matrix of the cost matrix in Table 2.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
</tr>
</tbody>
</table>

For example, consider a TSP problem starting from city A, visiting remaining cities once, and coming back to A where the cost matrix for those cities is shown in Table 4. A search tree produced by the A* algorithm is shown in Figure 1. The root node of the tree is the reduced matrix shown in Table 3, and the estimated lower bound of the cost for any tour starting from A is 20. From here we can travel to city B, C, D, or E, and the costs for to those traveling are 0, 5, 0, 4. These costs can be found on the first row of the root node. Each choice of cities introduces a new node, and the root node is appended to the list CLOSED. We name the nodes with the sequence of visited cities. Then, the child nodes of the root will be named AB, AC, AD, and AE, respectively. These choices and costs are written on
the edges, from the root node to the child nodes. Consider the node named AB, and let us find how to come up with the matrix. Since this node represents arrival at city B from A, we should never leave city A or visit B. We put at every entry of row A (the first row) to prevent leaving from A again, and at every entry of column B (the second column) to prevent revisiting city B. We also put at B-row-A-column entry, to prevent revisiting A from B. For the other entries, we copy from the root. Then, we reduce it by subtracting 1 from every entry of the first column. For each child of the root, we can find a number written just above it. For each node, nodei, we denote the name of nodei Stringi. The number written above nodei represents the estimated lower bound of the cost for the tour, whose initial part is stringi. For example, a lower bound for the tour whose initial part is “A-B” is 20+0+1=21, where 20 is the lower bound found for the parent of AB, 0 is the cost for traveling from A to B, which is also A-row-B-column entry of the root, and 1 is the sum of subtracted values for reduction.

In the same manner, the first rows of nodes AC, AD, and AE are all filled with . The column C of AC, column D of AD, column E of AE, C-row-A-column entry of AC, D-row-A-column entry of AD, and E-row-A-column entry of AE are also filled with . The estimated cost for node AC is 20+5+2=27, AD is 20+0+1=21, and AE is 20+4+1 = 25. The estimated minimum cost is 21. We select a node with an estimated cost of 21, AB, and expand it in the same manner as we have done with the root. Expanding AB introduces ABC, ABD, and ABE. Estimated costs for these nodes are 30, 27, and 21 respectively. At this moment, nodes in the OPEN list are AC, AD, AE, ABC, ABD, and ABE. The node with the minimum estimated cost is ABE. Thus, it will expand ABE. Expanding ABE introduces ABEC and ABED. Then, we expand ABEC. After this expansion, we can find the solution to the problem, ABECDA.
\begin{figure}
\centering
\includegraphics[width=\textwidth]{search_space.png}
\caption{The search space built by A*}
\end{figure}

\begin{table}
\begin{tabular}{|c|c|c|c|}
\hline
A & B & C & D \\
\hline
\hline
0 & 1 & 0 & 6 \\
\hline
\hline
20 & 20+0+1=21 & 20+5+2=27 & E, 4 \\
\hline
\hline
\hline
\hline
21+5+4=30 & 21+2+4=27 & 21+0+0=21 & \\
\hline
\hline
\hline
\hline
C, 5 & D, 2 & E, 0 & \\
\hline
\hline
\hline
\hline
21+0+0=21 & 21+6+56=32 & \\
\hline
\hline
\hline
\end{tabular}
\end{table}
III. PROPOSED METHODS

The purpose of our homepage is to provide an optimal route to a user who wants to travel in Korea. For this purpose, we need both the shortest path, and its cost for every pair of cities in Korea. We can obtain this data by making use of Warshall's algorithm. However, the algorithm only produces minimum costs, not the shortest paths. To find both costs and paths, we slightly modified Warshall's algorithm. Our modified version of Warshall's algorithm is shown in Table 6. We keep the shortest path from city I to city j at Path[I, j] for all the cities. Path[I, j] is initially city i if [I, j] entry of the adjacency matrix is not , nil otherwise. This means that at the initial path from city i to city j consists of city i if there is a direct road from city i to city j, and there is not a path from city i to city j if there is not a direct road from city i to city j. Whenever A[I, j] is changed into A[I, k]+A[k, j], Path[I, j] is also changed into concatenation of Path[I, k] and Path[k, j]. Therefore, whenever the cost of the shortest path from city i to city j is changed, the shortest path is recorded at Path[I, j].

<Table 6> Our Algorithm to find all the shortest paths

/* W(n, n) is the weighted adjacency matrix of a graph with n vertices:
   A(i, j) is the cost of a shortest path from vi, to vj.
   Path[i, j] is the shortest path from vi, to vj.
   W(i, i) = 0, 1<i<n */

procedure Find_PATHS(W, A, n)
integer I, j, k, n; real W(n, n), A(n, n)
for I := 1 to n do
   for j := 1 to n do
      A(i, j) := W(i, j)
      If W(I, j) == then Path[I, j] = Nil;
         Else Path[I, j] := City i
      repeat
   repeat
   for k := 1 to n do
      for I := 1 to n do
         for j := 1 to n do
            if W(I, j) > W(I, k) + W(k, j) then
               Path[I, j] := City k
               W(I, j) := W(I, k) + W(k, j)
            else
               if W(I, j) == W(I, k) + W(k, j) then
                  Path[I, j] := Concatenate(Path[I, k], Path[k, j])
            end if
         end if
      end for
   end for
end procedure

for i := 1 to n do
for j := 1 to n do
    if \( A(i, j) > A(i, k) + A(k, j) \)
        \( A(i, j) \leftarrow A(i, k) + A(k, j) \)
        Path[i, j] := concat(Path[i, k], Path[k, j])
repeat
repeat
repeat
end Find_PATHS

The existing approach for TSP is to use \( A^* \) algorithm equipped with the heuristic function mentioned in Section II, assuming that the first city is the same as the last city, and that the user is coming back to his hometown after traveling. This is often not the case. Suppose a foreigner wants to enter Korea through Kimpo airport and leave through Kimhae airport after visiting some cities; we cannot use the existing approach for TSP in this case. However, our homepage can handle this kind of problem by representing the first city and the last city on the cost matrix, and retaining the branch for visiting the last city during the expansion. For example, consider the example introduced in Section II. Without loss of generality, let A be the first city and E be the last city. What the user wants is to not return to city A, and never leave from city E. This can be represented by the 's filling the column A and row E of the cost matrix as shown in Table 7. The reduced matrix of Table 7 is shown in Table 8, and the cost for the reduction is \( 3 + 6 + 2 + 2 + 0 + 0 + 1 + 0 + 0 + 1 = 15 \). Let table 8 be the root node, and proceeds the \( A^* \) algorithm as the tree shown in Figure 1. Then, we obtain the search tree as shown in Figure 2. Look at the branches from the root node. The branch for visiting city E is not shown. This is because E is supposed to be visited last. The remainder of the tree is the same as the tree shown in Figure 1. The answer for this problem is "A-B-D-C-E", and the cost for this travel is 17.
Fig. 2 A search space for the problem where the start and the destination are different.
<Table 7> The cost matrix obtained from Table 4 by representing the first city (city A) and the last city (city E).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>∞</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>9</td>
<td>∞</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>6</td>
<td>2</td>
<td>∞</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

<Table 8> The reduced matrix of the cost matrix shown in Table 7.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>6</td>
<td>∞</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>3</td>
<td>0</td>
<td>∞</td>
<td>8</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

IV. IMPLEMENTATION

There are 170 cities and towns in Korea. Therefore, we need a 170x170 matrix representing the shortest paths, and the costs between all the pairs of these cities. However, obtaining the entries of the matrix by using a map is too time consuming. Instead, we obtained the distances for each city to its neighbor cities. This data is shown in Table 9, and is the input data to our system. Our system runs the modified Warshall’s algorithm shown in Table 6 on this data. The result of this step is the cost matrix, C, for 170 cities in Korea.

<Table 9> List of all the cities in Korea and the distances between neighbor cities.

Kyungjoo
Ulsan 41, Pohang 31, Youngchun 34, Yangsan 59, Milyang 79.2, Chungdo 66.8

- 348 -
Pohang
Kyuungjoo 31, Youngchun 46, Ulsan 88, Youngduk 44, Chungsong 73

Youngduk
Pohang 44, Uljin 72.5, Chungsong 55, Youngyang 53, Ahndong 77.5

Uljin
Youngduk 72.5, Bongwha 88.5, Ahndong 116.5, Samchuck 61, Taeback 102.5,
Youngyang 108, Youngwuel 175.5

Samchuck
Uljin 61, Taeback 37.5, Donghae 11.5, Jungsun 70.5

Donghae
Samchuck 11.5, Kangreung 40, Jungsun 70

Kangreung
Donghae 40, Yangyang 46.5, Pyungchang 99.5, Wonjoo 131, Hoengsung 127, Hongchun 139,
Choonchun 156.5, Inje 145.5

Yangyang
Kangreung 46.5, Sogcho 19, Inje 56, Yanggu 78

..., and so on

Our home page displays a map of the southern half of the Korean Peninsula, with prompts asking for start city, last city, and other cities to be visited, and finds the optimal path to display on the map. Since the path should be displayed dynamically, our system should be implemented in JAVA language.

Whenever invoked, JAVA executes init() first, then it executes start(), and paint() concurrently. We therefore display the prompts in the init(). Using get Image(), the init() also loads the image file of the map and associates it with a handle, img, so that paint() can draw the map using the handle. JAVA provides Choice object and Check box object. Choice
object can have a list of several choices, and asks a user to select one. Choice Test( ) in the init( ) instantiates Choice objects twice, one for Start City and the other for Destination, and puts them into a panel, then returns the panel so that the init( ) displays the panel at the top (north) of the screen. JAVA also provides Check box object. This can have a list of many items, and asks the user to select as many as he wants. We are instantiating Check box once, and using it for the user’s selection of the cities that he wants to visit. Check box Test( ) instantiates a check box filled with all the names of cities in Korea, and returns the check box so that the init( ) displays it at the right side (East) of the screen.

![Diagram showing the initial screen of the Java applet.]

<Fig. 3> The initial screen.
<Table 10> A part of init( )

...  
Image img = getImage(getDocumentBase( ),"jido.gif");  
...  
setLayout(new BorderLayout( ));  
add("South", p2);  
add("North", choiceTest( ));  
add("East", CheckboxTest( ));  
...  

A JAVA program flows as events occur. Examples of events are clicking the mouse button on a choice list, a check list or a button, time elapse, typing keyboard, etc. These events are handled in action( ). Action( ) invokes an appropriate function as an event occurs. A part of action( ) is shown in Table 11.

<Table 11> A part of action( ).

public boolean action(Event evt, Object arg)

    if(!(evt.target instanceof Checkbox) & & !(evt.target instanceof Button) & & !(evt.target instanceof Choice))
        return false;

else if(evt.target instanceof Choice)
    ... // handling choice button selection

else if(evt.target instanceof Button)
    ... // invokes the A* algorithm

return true;

- 351 -
<Fig. 4> Cities to be visited have been selected

V. EXPERIMENTS

The home page's address is http://wwwwcs.dongruk.ac.kr/~vim. Select D3CSL, then "Current Research Topic", and then "Optimal Path Algorithm", to see the screen as shown in Figure 3. If you select START, a list of several cities with an international airport nearby appears so that you can select one. In the same manner, you can select the last city by pressing DESTINATION next to START. Then, you choose cities to visit from the checklist. Figure 4 is the screen when a user has made his selections for cities to be visited. This figure shows that 36 Namwon, 47 Mokpo, 102 Ulsan, 124 Jindo, and 145 Pohang have been selected. Figure 5 is the screen which shows the optimal tour for the user.
VI. CONCLUSIONS

This paper introduced our homepage, that finds an optimal tour for a user when he selects some cities he wants to visit. Our homepage has been implemented in JAVA language, so that the suggested tour can be dynamically drawn on the map. To find the shortest path between every pair of cities in Korea, we have modified Warshall’s algorithm. We also modified A* algorithm for TSP (Traveling Sales Person), so the first city may be different from the last city.

<Fig. 5> An optimal path is shown.
REFERENCES


Seo, Taiyang (1996), Introduction to Tourism, Bummoone Publishing Co.

http://www.kvongbuktour.or.kr
http://www.bulguksa.or.kr
http://www.w3.org/pub/WWW/MarkUp/MarkUp.html