韓國 컴퓨터情報學會 論文誌
第12卷 第5號, 2007. 11.

2007-12-5-1-3

# 여행지 최적 경로를 제공하는 웹 시스템의 설계와 구현

임 재 걸*, 이 강 재**

# Design and Implementation of a Web System Providing Optimal Travel Routes

Jaegeol Yim*, Kangjai Lee**

## 요 약

본 논문은 일반인들이 여행 시 필요한 최적의 경로를 찾아 주는 웹 사이트를 구현하는 방법을 제안한다. 출발지와 목적지만 주어질 때 최적 경로를 찾아 주는 웹 사이트는 이미 많이 존재한다. 그러나 방문할 도시가 여럿일 때 최적 경로를 찾아주는 사이트는 아직까지 존재하지 않는다. 출발 도시에서 출발하여 경유지를 모두 한 번씩 방문하고 출발도시로 되돌아오는 최적 경로를 찾는 문제를 외판원 문제라고 하며, 이 문제는 지수 시간복잡도 문제로 널리 알려져 있다. 본 논문에서는 인공지능 탐색 알고리즘을 적용하여 외판원 문제를 푸는 방법을 웹 시스템으로 구현하였다. 지금까지 소개된 외판원 문제를 푸는 알고리즘은 출발지와 도착지가 동일한데 반하여 최적 경로 문제에서는 출발지와 도착지가 서로 다를 수도 있다. 본 논문이 제안하는 방법은 출발지와 도착지가 동일하거나 아니면 다르더라도 최적의 경로를 찾는다는 점이 기존의 연구와는 다르다. 본 논문에서 구현한 웹 사이트는 사용자에게 출발지, 목적지, 그리고 여러 경유지들을 선택하게 한 다음, 출발지를 출발하여 모든 경유지들을 경유하고 도착지로 도착하는데 비용이 가장 적게 드는 방문 순서를 효율적으로 찾아준다.

## Abstract

We have implemented a WWW homepage which finds an optimal route for users. There already exist many web sites which provide the optimal route when a start and a destination cities are given. However, none of them can find the optimal route when a number of cities to be visited. The problem of finding the optimal route starting at a given start city and visiting through all the given intermediate cities and finally returning to the start city is called Travelling Sales Person(TSP) problem. TSP is a well known exponential time complexity problem. We have implemented an artificial intelligent search algorithm for TSP on our homepage. The main feature of our algorithm is that the destination may not be the same as

• 제1저자 : 임재걸 • 교신저자 : 이강재
• 접수일 : 2007. 8.29, 심사일 : 2007. 9.1, 심사완료일 : 2007. 10.21.
* 동국대학교(경주) 컴퓨터멀티미디어학과 교수, ** 수원과학대학 컴퓨터정보과 교수

the start city whereas all of the existing heuristic algorithms for TSP assume that the start and the destination cities are the same. The web page asks a user to select all the cities he or she wants to visit(including start and destination city), then it finds a sequence of the cities such that the user would travel minimum distance if he or she visits the cities in the order of the sequence. This paper presents algorithms used in the homepage.

▸ Keyword : the optimal route, Travelling Sales Person(TSP) problem, minimum distance, the web page

## I. 서 론

지도 서비스를 제공하는 인터넷 사이트는 나라마다 수 십 개가 넘는다. 우리나라의 지도 서비스 제공 사이트도 naver 등을 비롯하여 10여 개가 넘는다. 이러한 사이트들은 관심 있는 지점의 지도상 위치를 찾아주고, 출발지에서 목적지로 가는 경로를 찾아주는 등 매우 유용한 정보를 제공한다[1-8].

그러나 기존의 지도 서비스 웹 시스템의 단점으로는 경유지를 고려한 최적 경로를 찾아주는 기능의 부재를 들 수 있다. 그래프 이론 분야에서 완전 유향그래프와 출발 정점 A가 주어질 때, A를 출발하여 다른 모든 정점을 꼭 한 번씩 방문하고 A로 되돌아오는 최적 경로를 찾는 문제를 일반적으로 외판원(TSP, Traveling Sales Person) 문제라고 한다. 외판원 문제는 잘 알려진 것처럼 지수복잡도 문제이다. 지도에서 경유지를 고려한 최적 경로를 찾는 문제 역시 그래프 이론의 외판원 문제와 같은 수준의 문제이기 때문에 지수복잡도 문제라고 할 수 있다.

어떤 시스템은 하나의 경유지 입력을 허락하기도 한다. 이 시스템의 경우에 경유지를 입력하면 출발지에서 입력된 경유지까지의 최적 경로를 찾고, 그 곳에서 도착지까지의 경로를 찾아준다. 또 어떤 사이트는 여러 개의 경유지를 입력하는 것을 허락한 다음, 입력된 순서대로 경유지들을 방문하는 경로를 찾아준다. 본 논문에서 설계하고 구현하는 웹 시스템은 입력된 경유지들의 방문 순서를 방문에 드는 비용이 최소가 되도록 결정한다는 점에서 기존의 시스템과 다른 새로운 최적 경로 제공 웹 시스템을 구현하고자 한다.

또한, 본 웹 시스템은 출발지와 목적지가 동일하여도 되고 서로 달라도 된다는 점에서 기존의 외판원 문제와는 차별화된다. 출발지와 목적지가 동일한 경우에는 기존의 외판원 문제를 푸는 A* 알고리즘을 그대로 적용하여 해를 찾지만, 서로 다른 경우의 문제에 대한 접근방법은 본 논문에서 새롭게 제시하고자 한다.

외판원 문제를 푸는 A* 알고리즘은 방문 도시들을 정점으로 하는 인접행렬을 입력데이터로 받는다. 본 웹 시스템은 전국 도시를 대상으로 하기 때문에, 방문도시들에 대한 인접행렬을 제공하려면 전국 시・군의 수가 150여개에 이르므로 22,500여 가지의 시・군 간의 거리를 알아야 한다. 그러나 22,500여 가지의 시・군 간의 거리를 측정한다는 것은 너무 시간과 노력이 많이 드는 작업이다. 본 논문에서는 이 문제를 해결하기 위하여, 지도에 나타나는 각 시・군에 대해 바로 이웃한 도시간의 국도의 거리만 측정하였고, 이 데이터로부터 Dijkstra 알고리즘[9]을 사용하여 모든 쌍의 도시간의 최단거리를 구하였다.

## II. 구현 알고리즘

### 1. 도시간의 최단 경로를 구하는 알고리즘

여행의 최적 경로를 구하려면 전국 시・군 간 국도의 거리가 필요하다. 전국 시・군의 수가 150여개에 이르므로 22,500여 경우의 수에 대하여 시・군 간의 거리를 측정하여야 한다. 이것을 지도에서 직접 찾아 기입한다는 것은 불가능한 일이다. 이 문제를 풀기 위하여 그래프 이론에서 개발된 "주어진 정점에서 다른 모든 정점까지의 최단 경로 찾기" 알고리즘을 사용하였다.

그래프는 정점의 집합, V와 V상의 관계인 E로 구성된다. E의 원소를 간선이라 한다. 각각의 간선에 실수가 연합되어 있는 그래프를 무게그래프라 한다. 정점과 간선의 나열 $v0\ e1\ v1\ e2\ v2\ \ldots\ ek\ vk$는 $ei$가 $(vi-1, vi)$인 간선일 때 $v0$에서 $vk$로 가는 경로라 하고 모든 $ei$의 무게의 합을 이 경로의 비용이라 한다. 무게그래프에서 이웃하는 정점간의 거리를 바탕으로 주어진 정점으로부터 다른 모든 정점 각각까지의 최단 거리 및 경로를 구하는 알고리즘은 Dijkstra 알고리즘을 비롯하여 여러 가지가 존재한다.

## 2. 최적 여행 경로 구하기 알고리즘

경유지들을 한 번씩 방문하고 출발지로 되돌아오는 최적 경로를 구하는 문제는 외판원 문제라는 이름으로 널리 알려진 대표적인 NP-complete 문제로써, 많은 사람들이 외판원 문제를 푸는 효율적인 방법을 찾아내기 위하여 노력하여 왔다. 가장 대표적인 외판원 문제를 접근하는 방법은 그래프 탐색 알고리즘의 하나인 A* 방법이다.

그래프 탐색이란 상태를 정점(node)으로 하고 합법적인 상태 변화를 간선으로 하며 상태 변화에 드는 비용을 간선의 무게로 하는 그래프에서, 초기 상태로부터 목적 상태에 도착하기 위한 최적 경로를 찾는 문제이다. 그래프 탐색 문제에서 상태의 수는 보통 기하급수로 증가한다. 외판원 문제에서 여행자가 현재 위치한 도시를 상태로 하고, 도시에서 다른 도시까지 직접 연결된 국도를 합법적인 상태 변화로 하면, 외판원 문제도 그래프 탐색 문제로 변환된다. 문제에 주어진 도시의 수가 n이라 하고, 어떤 도시에 이웃한 도시의 수를 평균 m이라 하면, 외판원 문제의 그래프 탐색 문제를 구성하는 상태의 수는 약 nm이 된다. 따라서 초기 상태로부터 모든 가능한 상태를 차례로 섭렵하면서 목적 상태에 도달하는 최적 경로를 찾는 방법은 도시의 수가 커짐에 따라 메모리 부족으로 수행이 불가능하게 된다.

이와 같이 상태의 수가 다룰 수 없을 만큼 많은 문제를 접근하는 방법으로, 각각의 상태에서 목적 상태에 도달하기까지의 비용을 예측하고, 가장 비용을 절약할 수 있으리란 상태를 선택하여 나아가는 방법을 알고리즘 A라고 하며 인공지능 분야에서 널리 사용된다. 더 나아가서 예측 비용이 언제나 실제 비용보다 작을 때, 이러한 예측 비용을 사용하는 알고리즘을 인공지능 분야에서는 A*라고 한다. 알고리즘 A*는 항상 최적의 해를 찾는다고 알려져 있다[10, 11, 12].

A* 알고리즘의 효율성을 결정하는 가장 중요한 요소는 예측치를 산출하는 방법의 정확성이다. 외판원 문제에서는 모든 도시를 꼭 한 번씩 방문하고 제자리로 돌아오기 때문에, 여행 경로는 방문지 각각에 대하여 한 번씩은 들어가고, 한 번씩은 떠난다. 따라서 여행에 드는 비용은 거리 행렬의 각 행의 최소값과 각 열의 최소값의 총합 보다는 작지 않다[13].

외판원 문제는 기본적으로 출발 도시와 종착 도시가 같다고 가정한다. 그러나 실생활에서는 출발지와 종착지가 다른 경우가 발생할 수 있다. 예를 들면 서울로 입국하여 렌트카를 빌려 관광을 하다가 김해 공항에서 출국하는 수가 있을 수 있다. 본 웹 시스템은 출발지와 종착지가 동일할 경우는 물론 다를 경우에도 최적 경로를 제공하여 준다. 전자의 경우에는 기존[13]의 방법을 그대로 적용한다. 후자의 경우에는 출발 도시로 들어올 수 없도록, 종착 도시에서 나갈 수 없도록, 그리고 여행 도중에 종착 도시로 들어가는 일이 없도록 미리 방지함으로써 최적 경로를 찾아 준다.

제안한 방법을 예를 들어 설명하면, 출발 도시 A, 종착 도시 E, 경유 도시 B, C, D, 각 도시간의 거리는 〈표 1〉의 비용행렬과 같다고 하자. 제안된 방법은 우선 〈표 2〉에 보이는 바와 같이 A열과 E행에 ∞를 넣어 A도시로 들어오는 경로나 E 도시에서 나가는 경로를 선택할 수 없도록 한다. 여기서 열은 들어오는 경로를, 행은 나가는 경로를 의미한다.

표 1. 다섯 개 도시의 거리 행렬
Table 1. Matrix representation of distances between cities.

| 도시 | A | B | C | D | E |
|---|---|---|---|---|---|
| A | ∞ | 4 | 8 | 3 | 10 |
| B | 4 | ∞ | 9 | 6 | 7 |
| C | 8 | 9 | ∞ | 2 | 5 |
| D | 3 | 6 | 2 | ∞ | 11 |
| E | 10 | 7 | 5 | 11 | ∞ |

표 2. 출발 도시와 종착 도시 표시
Table 2. Representation of Start city and Destination city

| 도시 | A | B | C | D | E |
|---|---|---|---|---|---|
| A | ∞ | 4 | 8 | 3 | 10 |
| B | ∞ | ∞ | 9 | 6 | 7 |
| C | ∞ | 9 | ∞ | 2 | 5 |
| D | ∞ | 6 | 2 | ∞ | 11 |
| E | ∞ | ∞ | ∞ | ∞ | ∞ |

모든 도시를 한 번씩 반드시 통과하는 경로의 총비용은 최소한 각 도시에서 다른 도시로 나가는 거리 중 최단 거리들과 각 도시로 들어오는 거리 중 최단 거리의 합보다 작지 않다. 즉, 도시 A에서 출발하여 다른 도시까지 가려면 최소 3 단위의 비용이 들고, 이 최소한의 비용을 사용하였다고 가정하면 도시 A에서 B, C, D, E까지 가는 거리로 각각 1, 5, 0, 7이 남는다. B에서 나가는 길의 거리는 최소 6, 이 비용을 사용하고 나면 B에서 C, D, E로 가는 비용은 각각 3, 0, 1이 남는다. C에서 나가는 길은 최소 비용이 2이고, 이 비용을 소비하고 나면 C에서 B, D, E로 가는 비용은 각각 7, 0, 3이 남는다. 마찬가지로 D에서 나가는 길은 최소 비용이 2이고, 이 비용을 소비하고 나면 B, C, E로 가는 비용은 각각 4, 0, 9가 남는다.

다음에는 각 도시로 들어오는 길의 최소 비용을 고려하여 보

자. 위의 과정을 마치고 나면 B열의 항은 각각 1, ∞, 7, 4, ∞로 B로 들어오는 길의 최소비용은 1이 되고, 이 비용을 소비하였다고 가정하면 남는 비용은 각각 0, ∞, 6, 3, ∞로 된다. 이 방법을 C, D, E에 적용하면 이들 도시로 들어가는 최소비용이 각각 최소 0, 0, 1인 것을 알 수 있다. 그러므로 〈표 2〉의 행렬에서 원하는 경로의 비용은 적어도 3 + 6 + 2 + 2 + 1 + 0 + 0+1 = 15 이상임을 알 수 있다. 이 비용을 사용하고 나면 각 비용은 〈그림 1〉의 근노드(노드 1)에 보는 바와 같이 각 행과 열에 0 항목이 있는 행렬로 변형된다. 이와 같이 각 행과 열의 최소항의 값을 각 항에서 빼는 과정을 축소라고 칭하고, 축소하기 위하여 뺀 값들의 합(위의 예에서는 15)을 축소비용이라 하며, 외판원 문제를 푸는 A* 알고리즘의 예측비용으로 사용한다.

도시 A에서 갈 수 있는 곳은 B, C, D, E 네 곳이지만 E는 반드시 다섯 번째 도시어야 함으로 제외됨으로, 노드 1의 자식노드는 세 개뿐이다. 첫 번 방문지로 B가 선정되었을 경우를 고려해보자. 노드 1에 나타난 도시 A와 B간의 거리는 0이다. 도시 A에서 B로 가는 경로가 선택되었음으로 A행과 B열에 ∞를 넣음으로써 또 다시 나가(들어오)는 경우를 선택하지 않도록 한다. 이렇게 얻은 행렬이 노드 2이다. B, C, D 행과 C, D, E 열에 0 항이 있으므로 이들 도시로 들어가거나 나가는 데 드는 최소비용의 총합은 0이다. 그러므로 노드 2의 경로를 선택한 경우 원하는 경로의 비용은 최소 15로 예상된다. 비슷한 방법으로 C나 D를 선택한 경우가 각각 노드 3과 4이며 이때의 최소비용은 각각 15 + 5 + 3 = 23과 15 + 0 + 5 =20이다.

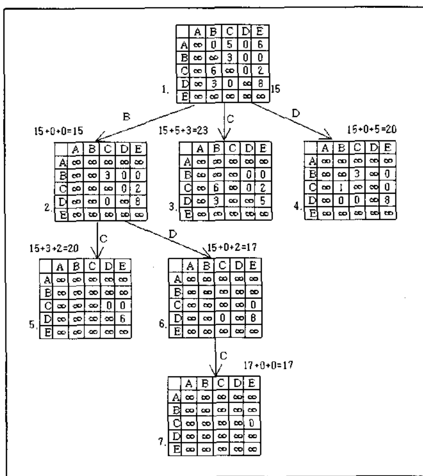



그림 1. 출발지가 A, 도착지가 E, 경유지가 B, C, D일 때 최적 경로를 찾는 알고리즘의 탐색 트리
Figure 1. Search tree of our algorithm for the example problem where Start is A, Destination is E and the others are visiting cities

예상되는 최소비용이 가장 작은 노드 2를 확장하면 노드 5와 6이 생성된다. 이때 역시 도시 E로 가는 선택은 제외한다. 노드 5와 6에서 예상되는 최소 비용은 각각 15 + 3 + 2 = 20과 15 + 0 + 2 = 17이다. 지금까지 열린 노드(확장되지 않은 노드)는 노드 3, 4, 5, 6이며, 이들 노드에서 예상되는 최소비용들 중 최소값은 17이다. 따라서 노드 6을 확장하면 예상되는 비용이 17인 노드 7을 생성하게 되며, 이것을 다시 확장하면 A, B, D, C, E의 순서로 여행하라는 경비가 17인 최적의 해를 구하게 된다.

## III. 성능 비교

여기서는 본 논문에서 구현한 웹 시스템에 적용한 알고리즘의 효율성을 억지기법(brute force : 방문순서를 모두 나열하고, 이들을 모두 비교하여 최적 경로를 구하는 방법)과 비교하고 그 실험 결과를 제시한다.

### 1. 억지기법

억지기법은 단순하게 문제를 해결하며, 모든 경우의 수를 다 비교하여 가장 작은 비용을 사용한 경우를 선택한다. 예를 들면, 4개의 도시 A, B, C, D, 출발도시 A, 도착도시 D가 주어졌을 경우, 경유지의 수가 2임으로, 억지 기법은 2! = 2 가지의 경우를 비교하여 여행 경비가 적은 경로를 문제의 해로 결정한다.

만약 도시가 5개(A, B, C, D, E), 출발지가 A, 도착지가 E, 나머지가 경유지라고 하자.

경우 1 : A -> B -> C -> D -> E
경우 2 : A -> B -> D -> C -> E
경우 3 : A -> C -> B -> D -> E
경우 4 : A -> C -> D -> B -> E
경우 5 : A -> D -> B -> C -> E
경우 6 : A -> D -> C -> B -> E

경유지의 수가 3임으로 위와 같이 3!가지의 경우가 생긴다. 3!개의 경우 중에서 여행 경비가 적은 쪽이 문제의 답이 된다. 이렇듯 방문할 도시의 개수가 정해진 상태에서 생각할 수 있는 모든 가능한 경우, 즉 (N-2)!개의 경우를 비교하여 가장 여행 경비가 적은 쪽을 선택하는 방법이 억지기법이다.

이 기법을 재귀 알고리즘으로 구현한 함수 travel은 〈표 3〉과 같다. 매개변수 path[]에는 선택된 도시들이 저장되며, 시작도시가 0번 cell에 그 다음 선택된 도시가 그 다음에 차례

대로 저장되며, path[N-1]은 도착도시가 된다. travel은 path에 삽입되어 있지 않은 모든 도시들 각각(v라 하자)에 대하여, path에 v를 추가하고, path에 N 개의 도시가 다 들어와 있으면 최적 경로와 최적 비용을 기록하는 배열 final[]과 finalcost를 갱신한 다음 path의 마지막 방(v가 들어 있음)을 0으로 바꾼 다음 종료(return)하며, path에 삽입되어 있는 도시의 수가 N이 아니면 v가 추가된 path를 매개변수로 travel을 재귀호출 한다. 예를 들면, 도시의 수가 4개(A, B, C, D), 출발도시가 A, 도착도시가 D인 경우, path[]에 A, B, C, D 순으로 저장 되어 있으면, 리턴 시 C가 0로 변한다.

표 3. 최적 경로를 찾는 억지기법 알고리즘
Table 3. Brute force algorithm to find an optimal path

```
조건 : 전역변수 matrix[i][j]는 도시 i에서 도시j로 직접
가는 비용, 인접행렬 final[]은 여행 경비가 가장 적은
경우의 경로 저장, finalcost는 여행 경비가 가장 적은
경우의 경비 저장
void travel(int path[], int cost, int count)
{
 1 check[]에 모든 도시의 이름을 차례대로 나열,
   여기에서는 도시의 이름이 숫자임으로 1부터 순서대로
   기억
  // check는 path에 고려된 도시를 기록할 배열임
 2 path[]에 저장된 도시의 이름을 찾아서 check[]에
   해당하는 도시를 0
  // path[] = {1, 2, 0, 0, 5} 이면, check[] = {0,
   0, 3, 4, 0}
 3 for( i=0; i < N-1; i++)   // N은 도시의 수
   {
  1 if (check[i] != 0)
   // path에 삽입되어 있지 않은 모든 도시 각각에
   대해...
    {
    3_1_1 이전 도시(path[i-1])에서 선택한
            도시(check[i])
          로 오는 값을 cost에 더하여 저장
    3_1_2 선택한 도시를 path[]에 추가함
    3_1_3 count 값을 1증가시킴
     // path[]에 도시 한 개가 저장되었기 때문에...
    3_1_4 if (count와 총 도시의 개수가 같으면)
          {
          1 최종 선택한 도시에서 처음 출발한 도시로의
            비용을 cost에 더함
          2 if (cost값이 finalcost값보다 작으면)
            {
            1 finalcost값을 cost값으로 바꿈
            2 final[]의 값을 path[] 값으로 바꿈
            }
           3 path[]의 마지막 값을 0으로 바꿈
           4 return;
          }
          else
           {
            1 travel(path[], cost, count) 재귀 호출
            2 다른 도시를 선택하기 위해서 증가시킨
               count
              값을 다시 감소 시켜 원래의 값으로 돌려놓음
              (3_1_3에서 증가 시킨 값)
            3 선택했던 도시가 저장된 path[]의 값을
               0으로
              바꿈(3_1_2에서 저장한 값을 의미)
            4 이전 도시에서 선택한 도시로 가는 값이
              3_1_1에서 cost변수에 반영되었는데 다시
              원래의 cost값으로 되돌림
           }
      }
   }
 4 return;
}
```

매개변수 cost에는 path가 나타내는 경로의 비용이 기록되어 있으며, 매개변수 count는 path[]에 삽입된 경유 도시의 수를 나타낸다. 즉, path에 경유 도시의 이름이 1개 저장되면 count=1이고, 2개 저장되면 2가 된다. count를 통해서 path[]에 경유 도시의 이름이 모두 저장되었는지 확인한다.

## 2. 출발지와 도착지가 다른 경우의 최적 경로를 찾는 A* 알고리즘

A* 알고리즘은 〈그림 1〉에 보이는 바와 같이 초기에는 근노드만 열린 노드로 하고, 열린 노드 중 예측비용이 가장 적은 노드를 선택한 다음 자식노드를 생성하여 열린 노드에 추가하는 작업을 모든 도시가 포함된 경로가 발견될 때까지 반복한다. 열린 노드들을 기록할 데이터 구조로 〈그림 2〉와 같은 Queue를 사용한다.

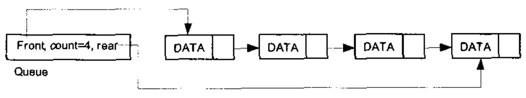



그림 2. 열린 노드를 기록할 데이터 구조
Figure 2. Our data structure for recording Open nodes

2차원배열 matrix는 〈그림 1〉에 보이는 바와 같은 각 노드의 행렬이다. 1차원배열 path[]는 각 노드에 이르는 경로이며, bound는 해당 matrix의 예측비용이다. 본 알고리즘은 열린 노드 중 bound가 가장 작은 노드를 선택하여 확장하는데, bound가 동일한 노드가 여럿이면 level이 가장 깊은 것을 선택한다. level까지 같은 노드가 여럿이면 먼저 생성된 노드를 선택하여 확장한다. 그래서 DATA에는 level 필드가 있고, 생성 순서를 나타내는 number라는 필드도 필요하다.

구현한 출발지와 도착지가 다른 경우의 최적 경로를 찾는 A* 알고리즘 travel_A는 <표 4>와 같다.

표 4. 출발지와 도착지가 다른 경우의 최적 경로를 찾는 A* 알고리즘
Table 4. A* algorithm to find an optimal path when Start and Destination are different

```
조건 : <그림 1>의 노드 1과 같이 출발도시와 도착도시가
표시된 비용행렬이 matrix 필드의 값인 DATA타입의
레코드를 초기 인수로 하여 호출함, 변수 count는 노드의
생성순서를 카운트한 것으로 노드의 번호(number)이다.

void travel_A(DATA leaf)
{
1 reduction함수를 호출하여 데이터 행렬의 값을 축소
2 queue를 생성
3 시작도시를 입력받고, 매개변수 leaf의 path[] 처음에 저
  장, path[]는 경로를 저장하는 변수
4 while( 1 )
  {
  1 leaf를 임시로 기억할 변수 temp2에 저장
  2 현재 어느 도시에 위치하고 있는지 leaf의 path[]에서
    찾고, 그 도시의 위치를 nowcity에 저장
  3 leaf의 비용행렬 matrix[][]에서 현재 도시가 위치하고
    있는 행의 값, 즉 가로줄을 무한대로 설정
  4 check[]에 도시의 이름을 차례대로 저장
  5 leaf의 path[]와 비교하여 현재까지 지나간 도시들을
    check[]에 0으로 표시
  6 for(choosecity = 0; choosecity < N;
    choosecity++)
   {
   1 if (check[choosecity]의 값이 0이 아닐 때)
     {
     1 leaf를 임시 저장 장소인 temp에 저장
     2 선택한 도시를 temp의 path에 저장
       // 저장위치는 nowcity 다음...
     3 nowcity에서 새로 선택한 choosecity로 가는 값을
       temp2에서 찾아서 temp의 bound에 저장
     4 choosecity에서 nowcity로 되돌아가는 값을
                         무한대
     5 선택한 도시(choosecity)에서 처음 출발 도시로 가
       는 값 역시 무한대로 설정
     6 temp를 reduction함수로 보내 비용행렬을 축소
     7 새로운 가지노드가 생겼기 때문에 temp의 level의
       값을 1증가
     8 새로운 노드가 생성되었기 때문에 temp의 number
       에 count 값을 저장하고 count를 증가
     9 queue에 temp를 삽입 // 유효노드이기 때문...
     }
   }
  7 searchQueue 함수를 통해 현재 queue에 있는 노드
                         중
    에서 가장 작은 bound 값의 노드를 leaf에 저장
  8 if (leaf의 path[]에 모든 도시들이 저장)
     1 while()문 종료   // 문제 해결
  9 deQueue 함수를 통해서 leaf를 queue에서 삭제
 }
5 해결한 문제의 해를 출력
}
```

매개변수 leaf는 초기에 주어진 비용행렬이다. 함수 reduction은 축소행렬과 축소비용을 계산하는 함수이며, 함수 searchQueue는 Queue에서 bound(예측 비용) 값이 가장 작은 노드를 찾는 함수이고, 함수 deQueue는 지정된 노드를 Queue에서 삭제하는 함수이다.

### 3. 성능 비교 실험

알고리즘의 성능을 비교하기 위하여 억지기법 알고리즘인 <표 3>을 travel로, 출발지와 도착지가 다른 경우에 최적 경로를 찾는 A* 알고리즘인 <표 4>는 travel_A로 구현하였다. 사용한 컴퓨터 모델은 ASUS A6VM, CPU 모델은 Pentium M 760 도선 2.0GHz이며 RAM은 1GByte, 266MHz이다.

<그림 3>은 실행시간에 대한 실험 결과로 크기가 5인 행렬부터 시작하여 크기가 15인 행렬까지는 억지기법도 해를 구할 수 있었지만, 크기가 16일 때 억지기법은 프로그램이 종료를 하지 않아 해를 구할 수 없었다. 한편 travel_A는 크기 15까지 해를 구하는데 걸리는 CPU 시간이 모두 0초였고, 크기가 16일 때 겨우 0.016초였다.

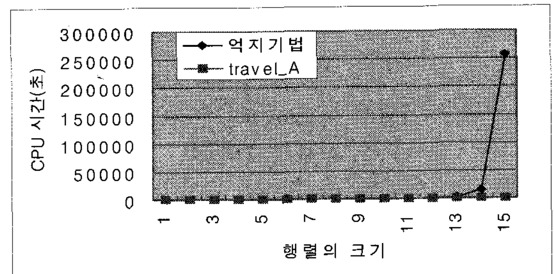



그림 3. 행렬의 크기에 따른 억지기법과 travel_A의 실행시간 비교
Figure 3. Comparison of execution times of brute force and travel_A considering size of matrix

이번에는 알고리즘 실행 중에 생성되는 노드의 수를 살펴보는 실험을 하였다. 실험 결과로 두 알고리즘이 생성하는 노드의 수를 비교하는 그래프는 <그림 3>과 대동소이하였다. 그래서 이번에는 travel_A가 생성하는 노드의 수만 <그림 4>와 같이 그래프로 나타내 보았다. <그림 4>에서 보는 바와 같이 travel_A도 역시 지수복잡도를 나타낸다는 것을 알 수 있었다. 결론적으로 travel_A는 <그림 3>에 보이는 바와 같이 억지기법보다 비교도 안될 만큼 더 효율적이지만, <그림 4>에서 보는 바와 같이 완만하지만 역시 지수복잡도임을 알 수 있었다.

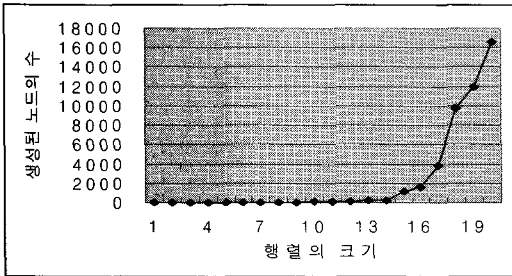



그림 4. 행렬의 크기에 따른 travel_A가 생성하는 노드의 수의 변화
Figure 4. Number of generated nodes vs. matrix sizes

# IV. 웹 시스템의 구현

본 논문에서 구현한 웹 시스템은 시・군 간의 거리 구하기와, 사용자의 출발 도시와 도착 도시, 그리고 방문지를 질의하고, 이들 간의 거리로부터 최적 경로를 구하여 이 경로를 동적으로 화면에 표현하여 주는 Java 프로그램으로 구현하였다.

## 1. 시・군 간의 거리 구하기

각 도시간의 거리를 구하기 위하여 우선 각 도시에 대한 이웃 도시까지의 거리를 지도에서 찾았다. 〈표 5〉는 이렇게 구한 거리의 일부이다. 〈표 5〉에서 보는 바와 같이 소수 첫째 자리까지 km를 단위로 구하였다. 거리 행렬을 실수형으로 하면 기억 장소를 너무 많이 필요로 함으로 해서 정수형으로 선언하고, 거리의 단위를 100m로 표시하였다.

Dijkstra 알고리즘에 〈표 5〉를 입력하여 각 시・군 간의 거리를 구할 수 있었다. 이 프로그램의 결과는 Java 프로그램의 인수로 사용된다. 그러나 이 결과는 시간에 따라 변화하는 것이 아니므로 굳이 Java로 구현되어야 할 필요가 없었다. 그래서 이 알고리즘은 C 언어로 구현하였다.

표 5. 이웃하는 도시간의 거리
Table 5. Distances between adjacent cities

| 도시 | 이웃 도시와의 거리(km) |
|---|---|
| 경주 | 울산 41, 포항 31, 영천 34, 양산 59, 밀양 79.2, 청도 66.8 |
| 포항 | 경주 31, 영천 46, 울산 88, 영덕 44, 청송 75 |
| 영덕 | 포항 44, 울진 72.5, 청송 55, 영양 53, 안동 77.5 |
| 울진 | 영덕 72.5, 봉화 88.5, 안동 116.5, 삼척 61, 태백 102.5, 영양 108, 영월 175.5 |
| 삼척 | 울진 61, 태백 37.5, 동해 11.5, 정선 70.5 |
| 동해 | 삼척 11.5, 강릉 40, 정선 70 |
| 강릉 | 동해 40, 양양 46.5, 평창 99.5, 원주 131, 횡성 127, 홍천 139, 춘천 156.5, 인제 145.5 |
| 양양 | 강릉 46.5, 속초 19, 인제 56, 양구 78 |
| 기타 | 이하 생략... |

## 2. Java 프로그램 구현

### 2.1 초기 화면

초기 화면에서는 한국 지도, 출발지, 목적지, 경유지들을 선택 할 수 있는 체크박스와 계산 결과를 보여 달라고 요구하는 버튼들을 보여준다. 애플릿은 우선 init를 호출하고, 계속해서 start와 paint 메시지를 비동기적으로 호출한다. 그럼으로 init에서 다음과 같은 명령으로 그림 파일을 image로 받고 이를 paint에서 화면에 출력시키도록 한다.

```
Image img = getImage(getDocumentBase(),
                "jido.gif");
```

여기서 getDocumentBase()는 현재 애플릿을 호출하는 HTML파일의 URL(Universal Resource Locator)을 return하는데 image가 HTML파일과 같은 URL에 있으므로 이 메서드를 사용한다.

### 2.2 최적 경로 구하기

먼저 paint() 메서드 내에서 화면출력 부분은 크게 아래와 같이 초기 화면 그리기와 계산 결과인 최적 경로를 그려주는 부분으로 나누어진다.

```
public void paint(Graphics g)
  {
   if (init_flag == 0){  ..................
     // 초기 또는 initialize 버튼이 눌려진 경우
      }
   else if (init_flag == 1) {  ..................
     // touring 버튼이 눌려진 경우
      }
  }
```

Java 프로그램은 사용자가 컴포넌트를 선택함에 따라 이를 처리하는 action 메서드가 작동함으로써 수행된다. instanceof는 어떤 컴포넌트 객체가 활동을 시작하였는지를 판별하여 준다.

```
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){
            ... // 초이스 버튼 선택 이벤트 처리
    }
    else if (evt.target instanceof Button){
            ... // 최적 경로 구하는 이벤트 처리
    }
  return true;
}
```

버튼, 메뉴와 같은 컴포넌트들은 레이블을 가지는데, 이 레이블로 이벤트 발생 객체를 참조할 수 있다. 레이블은 이벤트 객체의 인스턴스 변수 arg에 지정된다. 따라서 equals() 메서드를 이용하여 비교하였다.

```
public boolean action(Event evt, Object arg)
{
 ........
 else if (evt.target instanceof Choice)
   //초이스 버튼에서 event가 발생했을 경우
 {
     if (evt.target.equals(choice1)){
       // 출발도시를 선택하는 경우
       ............
   }
   else if (evt.target.equals(choice2)){
       // 도착도시를 선택하는 경우
       ............
   }
```

초이스 버튼과 체크박스에서 선택된 값은 keyPath[]에 저장 되어 있다. 이를 이용해 Dijkstra 알고리즘을 적용한 결과, 만들어진 전체 150여개 도시 간 최단 거리가 저장되어진 157×157 행렬인 matrix[][]에서 해당되는 행을 먼저 차례대로 고른 다음에 선택된 행에서 keyPath[]에 해당하는 열을 다시 선택한다. 이렇게 해서 선택된 출발지와 경유지, 그리고 도착지의 거리 데이터가 만들어지게 된다.

이때 출발지와 목적지가 다를 경우 출발지에서 들어오는 값과 도착지에서 나가는 값을 없애기 위해, 해당 출발지의 열과 해당 목적지의 행에 -1 값을 넣어준다.

## V. 실험 결과

<그림 5>는 본 논문에서 구현한 Applet WWW 페이지 http://wwwcs.dongguk.ac.kr/~yim/tsp2/tsp.html을 UNIX에서 Netscape 웹브라우저를 이용해 불러들인 후 초기화가 끝난 모습이다. 왼쪽에는 우리나라 지도가 보이고 오른쪽에는 모든 도시의 이름들이 나열되어 있다.

이 Applet을 사용하기 위해 먼저 Applet의 최상단에 위치한 버튼 중 왼쪽의 START 초이스 버튼을 누르면 도시 이름들이 나타나는데 그 중에서 출발지를 선택한다. 비슷한 방법으로 목적지는 START 옆의 DESTINATION이라고 레이블된 초이스 버튼을 클릭하여 선택한다. 실험을 위하여 여기서는 출발지를 서울로, 목적지는 부산을 선택하였다. 오른쪽에는 157개의 도시들이 GridLayout으로 배치되어 있는데, 그 중에서 방문할 도시들을 두 곳 이상 선택한다.

<그림 6>은 모든 방문도시에 대한 설정이 끝난 후에 최종 결과를 보기 위해 Touring 버튼을 누른 결과를 보여주는 그림이다.

여기서 경유지로 선택된 36 Namwon(남원), 47 Mokpo(목포), 102 Ulsan(울산), 124 Jindo(진도), 그리고 145 Pohang(포항)에 대한 최적의 경로로 선택된 도시들이 선으로 연결되어 있음을 볼 수 있다. 따라서, 이를 통해 서울을 출발하여 목포, 진도, 남원, 포항, 울산을 차례로 거쳐 부산으로 가는 것이 최적의 경로임을 알 수 있었다.

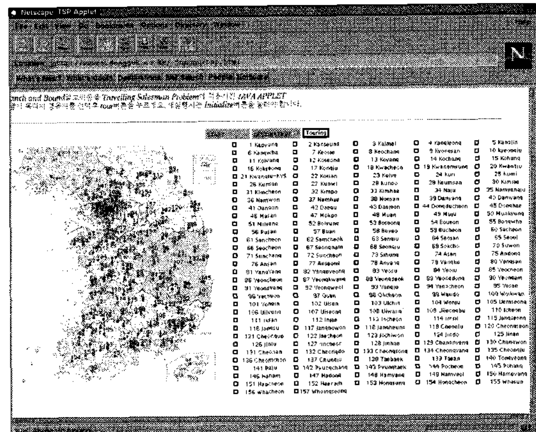


그림 5. Applet 초기화면
Figure 5. The Applet initial screen

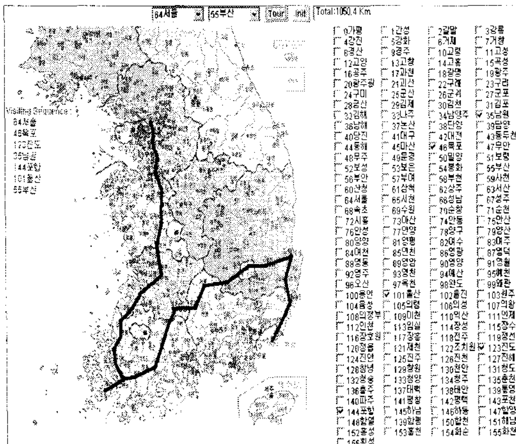


그림 6. 최적의 경로가 표시된 화면
Figure 6. An example showing an optimal path

## Ⅵ. 결 론

본 논문은 출발지와 종착지, 그리고 중간에 방문할 도시들을 선택하면 자동차 등으로 국도를 이용하여 가장 짧은 거리를 달려 선택된 도시들을 모두 한 번씩 방문하고 종착지에 도착하는 최적의 경로를 발견하는 알고리즘을 제안하고, 알고리즘을 구현한 웹 시스템을 구축하였다. 웹 시스템 구축을 위하여 우선 각 도시에 대한 이웃 도시까지 국도의 거리를 모두 조사하였고, 이 거리 데이터에 두 정점간의 최단경로를 찾는 알고리즘을 적용하여 모든 도시간의 최단 경로를 구하였다. 그리고 제안하는 알고리즘을 Java 애플릿으로 구현하였다.

본 논문에서는 여행하는 국도의 거리라는 관점에서 최적의 경로를 찾는 웹 시스템을 구축하였으며, 본 논문이 제안한 방문지 경유 최단 경로 찾기 알고리즘은 비용이 거리이든, 시간이든, 아니면 여비이든 어떤 조건에서도 무관하게 적용이 가능하다.

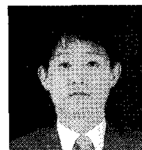
추후에는, 여러 가지 관점에서 비용을 최소화하는 최적의 경로를 찾아주도록 본 시스템을 개선하고자 하며, 나아가서 지리정보 기술을 적용하여 더욱 실용적인 시스템으로 개선하는 연구를 계속하고자 한다.

## 참고문헌


[1] http://maps.naver.com/
[2] http://local.paran.com/map/
[3] http://www.congnamul.com/
[4] http://kr.gugi.yahoo.com/ymap/map.php
[5] http://www.cybermap.co.kr/
[6] http://www.maptopia.com/
[7] http://maps.google.com/
[8] http://www.mapquest.com/
[9] Skvarcius and Robinson, Discrete Mathematics with Computer Science Applications, Reading, The Benjamin/Cummings Publishing Company, Inc. 1980.
[10] Gelperin, D. "On the Optimality of A*," Artificial Intelligence, 8(1), 1977, pp69-76.
[11] Hart, P.E., Nilsson, N.J., and Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", IEEE Trans. Sys. Science and Cybernetics, SSC-4(2), 1968, pp. 100-107.
[12] Nils J. Nilsson, Principles of Artificial Intelligence, Reading, Tioga Publishing Company. 1980.
[13] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Reading, Computer Science Press, 1984.
[14] http://wwwcs.dongguk.ac.kr/~yim/tsp2/tsp.html


## 저 자 소 개

임재걸
1981년 동국대학교 전자계산학과 학사
1987년 University of Illinois 석사
1990년 University of Illinois 박사
1992년 - 현재 동국대학교(경주)컴퓨터멀티미디어학과 교수
〈관심분야〉 멀티미디어시스템, 컴퓨터시스템 분석, 위치기반 서비스



이강재
1981년 동국대학교 전자계산학과 학사
1983년 동국대학교 대학원 전자계산학과 석사
1997년 동국대학교 대학원 컴퓨터공학과 박사
1986년 - 현재 수원과학대학 컴퓨터정보과 교수
〈관심분야〉 데이터베이스 응용, 데이터마이닝, 멀티미디어시스템