2005년도 대한전기학회 하계학술대회 논문집 2005. 7. 18 - 20

항법서버의 샛길 탐색 기능 구현

박원장*, 남승욱**, 이영일*
서울산업대학교 제어계측공학과*, (주)Witcom**

Implementation of a byroad searching system on a navigation server

Won Jang Park*, Seung Wook Nam**, Young Il Lee*
Seoul National Univ.,* Witcom Co., Ltd,,**

Abstract - 본 논문에서는 네비게이션 시스템 사용 시 교통 체증 등의 상황이 발생하여 최적 경로를 따라 가는 경우의 소요시간이 실제 예상시간보다 크게 증가하게 되는 경우 주변의 샛길을 이용하여 보다 빠른 시간에 체증구간을 회피하여 목적지에 도착 할 수 있게 하는 알고리즘을 제안하고 이를 항법서버에 구현한다. 교통 체증상황이 발생하는 경우 경로 탐색시의 각 길에 대한 weighting 을 조절함으로써 원래 최적경로와 다른 우회로를 찾아 낼 수 있도록 한다.

상용화 되어있는 네비게이션 단말기를 사용하였으며 PC 시뮬레이터와 개발 서버를 이용하여 샛길 탐색 기능을 구현하였다. 최적경로 탐색에는 A* 알고리즘을 응용하였으며 최적경로에서 샛길 탐색구간을 구하여 전체 경로를 탐색하는 알고리즘을 구현하였다.

1. 서 론

최근 차량과 운전자에게 다양한 정보 및 서비스를 제공하는 텔레매틱스와 사람이나 사물의 위치를 정확하게 파악하고 이를 활용하는 위치기반 서비스가 폭발적으로 증가하고 있다. 그중에서도 위치기반 서비스는 자동차용 navigator의 사용이 일반화 되어가고 있는 추세이다. 이러한 navigator 서비스에 교통정보를 반영한 서비스를 적용하게 되면 교통정보를 이용하여 체증구간을 회피해야 할 상황이 발생하게 된다. 현재 경로탐색은 각 도로를 링크단위로 나누고 각각의 링크에 대한 weighting 값을 평가하여 경로탐색 알고리즘을 통해 링크들을 출발지부터 목적지까지 연결하여 찾아내는 방법을 사용하고 있다. weighting 값은 각 링크에 도로종별, 차선 수, 링크종별, 속성종별, 회전 cost 등의 값을 조합하여 사용하며 weighting 값이 낮 을 수 록 경로탐색 시 선택 될 가능성이 높아진다. 링크들을 연결하는 방법에 있어 주로 Dijkstra algorithm[1] 을 응용한 A* algorithm[2] 을 많이 사용하고 있으며 A* algorithm[2] 은 Dijkstra algorithm[1]에 방향성 같은 경험에 의한 heuristic value 를 고려한 방식이다.

본 논문에서는 위치기반 서비스를 이용한 경로 탐색 서비스에 있어서 체증이 발생한 구간에서 우회경로를 탐색하는 방법을 제안한다. 이를 위해 현재 상용화 되어 있는 네비게이션 시스템[3]과 경로 탐색 수행 시 사용되는 알고리즘을 분석하고 위치기반 서비스의 경로탐색 수행에 가장 많이 사용되는 A* algorithm 을 응용하여 우회경로 탐색에 적용시키는 방법을 연구한다. 우회경로를 찾기 위해 각 도로의 차선 수를 판단하여 차선 수가 적을수록 weighting 값을 작게 만들어 샛길을 우선 시 하여 경로 탐색을 수행하며, 단순히 샛길을 최우선으로 하였을 때 나타나는 경로가 원 경로와 크게 벗어나는 상황을 방지하기 위해 체증구간이 시작되는 위치로부터 일정한 거리가 떨어진 원 경로상의 한 지점을 새로운 목적지로 설정하여 체증구간에서는 샛길 탐색을 하고 나머지구간은 원 최적경로를 사용하여 전체 경로를 구하는 방식을 사용한다. 또한 이 방법에서 샛길 구간과 최적경로 구간을 구분 할 때 현 출발지의 도로 속성을 판단하여 출발 지점의 위치가 고속도로 등의 주변에 샛길이 많지 않은 경우와 복잡한 시내 등의 주변 샛길이 많은 경우의 두 가지로 나누어 주변 샛길이 많고 원 경로상의 새로운 목적지까지의 거리가 짧은 경우는 샛길의 성질을 강조하기 위해 회전 cost 반영 비를 작게 만들어 회전에 대해 자유롭게 만들고 반대로 새로운 목적지가 멀고 샛길이 주변에 별로 없는 경우엔 회전 cost 반영 비를 높여 크게 우회하더라도 회전비율을 낮춰 원 경로의 경로탐색 결과와 큰 거리차이가 없도록 만든다. 본 논문의 2장에서는 기존 경로 탐색 방식에 대해 논하며, 3장에서는 기존 경로 탐색 방식을 응용하여 샛길 탐색 알고리즘과 방법을 설명한다. 4장에서는 샛길 탐색 알고리즘을 이용하여 서울시를 중심으로 실제 경로 탐색을 예로 실험 결과 및 추후 개선 사항에 대해 논한다.

2. 본 론

2.1 기존 경로탐색 알고리즘

2.1.1 weighting 계산방법

전체 경로를 링크단위로 나누고 빠른 길을 찾기 위해서 여러 가지 링크의 속성들을 고려해 각 링크의 weighting 값을 평가한다. 현 링크의 속성뿐만 아니라 실제 경로탐색 시에는 이전 링크와 현 링크, 현 링크와 다음 링크와의 관계 또한 고려대상이 된다. 기본적으로 각 링크에 weighting값을 평가하고 이를 비교해 나가면서 작은 값들을 중심으로 경로를 탐색한다. 평가된 weighting 값을 cost라 칭하며

cost = BaseCose*weighting1+weighting2 (1)

TotalCost=(CurrentCost+NextCost)/2+TurnWeight (2)

로 표현된다.

CurrentCost 와 NextCost는 각 링크의 BaseCost, 도로속성, 링크속성, 차선 수 등으로 평가된 값이다.

BaseCost = Length / VelocityLimit (3)

이 되며, 도로속성은 고속도로 또는 국도 등의 속성이며, 링크속성은 현 링크의 주변링크와의 연결 상태(Ramp, 로터리, 교차점내 링크 등)를 나타낸다. TurnWeight는 현 링크와 다음링크의 진행방향에 따른 회전 값이 되며 진행방향의 각이 클수록 큰 값을 가지며 좌회전보다 우회전이 작은 값을 갖는다.

2.1.2 Dijkstra 와 A* algorithm

평가된 weighting 값들의 집합을 이용하여 실제 경로를 탐색하기 위해서는 한 노드를 탐색 할 때마다 주변의 모든 노드를 비교해야하며 출발지로부터 목적지까지를 보면 지도상의 모든 노드를 비교해야 하기 때문에 노드의 수에 따라 시간 복잡 도는 크게 상승하게 된다. Dijkstra Algorithm은 출발지로부터 목적지가 나올 때까지 모든 연결 된 노드를 조사하는 방법이다. Dijkstra Algorithm에는 두 개의 집합 P, T 가 존재하며 처음 모든 노드는 집합 T에 속하며 집합 P는 빈 집합이 된다. 시작 노드에서 연결 된 모든 노드의 cost를 조사하고 cost가 가장 작은 노드를 선택한다. 선택 된 노드는 집합 P로 옮겨지고 다시 그 노드에서 집합 T에 있는 모든 노드를 조사한다. 집합 P가 모두 집합 T로 옮겨질 때까지 위의 과정을 반복 하며 이때 시간 복잡 도는 $O(n^2)$이 된다. Dijkstra Algorithm은 노드 수가 많은 지도에서는 많은 연산과정이 필요하다. 특히 현재 서비스 중인 네비게이션의 지도에 노드 수만 하더라도 약 2만개에 이르고 있고 계속 그 수가 증가하고 있다. 그래서 시간 복잡 도를 줄이기 위해 Dijkstra Algorithm에 Binomial 나 Fibonacci heaps 등을 통해 성능 개선을 할 수 있다. 개선된 Algorithm에는 Bellman-Ford Algorithm[4], Floyd-Warshall Algorithm등의 많은 Algorithm이 있으나 가장 많이 사용되고 있는 Algorithm이 A* Algorithm이다.

A* algorithm은 경로 탐색을 수행하는데 있어 매우 효과적인 알고리즘이며 다양한 종류의 문제들을 해결하는데 사용되어 왔다. A* algorithm은 출발지점에서 목표지점까지 가장 비용이 낮은(보통 가장 짧은) 경로를 찾는데, 다음과 같은

f = g + h 계산 값을 사용하여 다음으로 이동할 경로를 결정하는 방식이다.

g는 goal로서, 시작노드로부터 이 노드까지 오는 데 드는 비용이다. 시작노드에서 이 위치까지 오는 경로들은 여러 개가 있을 수 있는데, 이 노드는 그 경로들 중 특정한 하나를 의미한다.

h는 heuristic으로, 이 노드에서 목표까지 가는데 드는 '추정된' 비용이다. 휴리스틱이란 '경험에 기초한 추측'을 뜻한다. 이것이 '추측된' 비용인 이유는 목표까지의 실제 비용을 아직은 알지 못하기 때문이다. 휴리스틱을 계산하는데 여러 가지 방법이 있을 수 있지만, 일반적으로 현재 위치에서 목표까지의 일직선 거리 값을 사용한다.

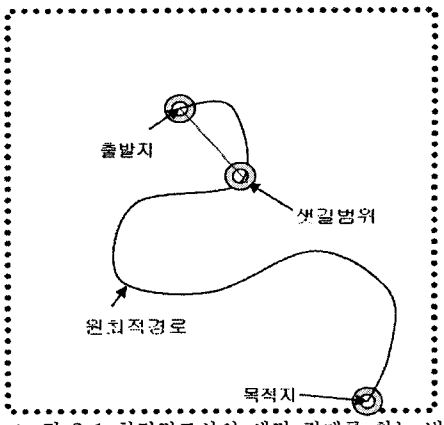
f는 fitness로서, g와 h의 합이고 이 노드를 거쳐 가는 경로의 비용에 대한 최선의 추측을 의미한다. f값이 낮을수록 이 경로가 최단 경로일 가능성이 크다.

현재위치에서 다음의 어떤 노드로 이동할 것인가를 결정하는 방법은 다음과 같다. 현재위치에 연결된 다음 노드가 여러 개일 경우, 각각의 다음 노드마다 시작위치에서 다음 노드까지의 거리와 그 노드에서 목표까지의 추정된 휴리스틱 값을 더한 f값들을 계산하여, f값이 가장 작은 노드 쪽으로 이동하는 것이 목표까지의 가장 짧은 경로가 될 가능성이 높다. 이렇게 f값이 가장 작은 노드들을 선택하여 탐색을 수행하다 목표노드에 도착하면 탐색은 종료하게된다..

2.2 샛길 탐색 algorithm

2.2.1 weighting을 이용한 샛길 탐색 방법

샛길의 경계를 정하는 방식은 직선을 잇는 방식이 아닌 최적경로를 먼저 구한 후 최적 경로 상에서 출발지로부터 누적된 거리가 입력된 샛길 범위와 일치하는 지점을 교차점으로 정하여 교차점을 새로운 목적지로 설정하여 탐색하는 방식이다. 그림으로 보면 다음과 같다.



<그림 2-1 최적경로상의 샛길 경계를 찾는 법>

처음 최적 경로를 먼저 구하고 구해진 최적경로의 링크를 출발점부터 하나씩 체크해 나가며 거리를 누적한다. 누적된 거리가 샛길 범위를 초과하는 순간의 링크를 샛길범위의 경계점으로 한다. 이렇게 경계점이 정해지면 출발지부터 경계점까지 샛길로 경로를 구한 뒤 경계점에서 목적지까지의 최적경로는 원래 경로를 구하던 루틴으로 넘겨서 계산한다. 이때 샛길 경로와 최적 경로를 합치는 방식은 기존에 경유지를 탐색하는 방식을 이용하여 샛길의 끝부분을 경유지로 인식시켜 합치는 방식을 사용한다.

샛길 cost계산 방식은 기존 cost계산을 위해 사용했던 도로종류 , 특수 도로가중치, 링크 종류, 속성종별에 대한 weight 비중을 상대적으로 낮게 평가하고 여기에 차선 수에 대한 비중만을 우선시 하는 방식이다. 또한 샛길은 주로 직선상으로 뻗어 있기 보단 구불구불하게 회전을 많이 하는 경우가 일반적이므로 이와 같은 현상을 방해하는 회전cost 또한 고려하지 않는다. 회전 cost는 가능하면 회전을 못하게 만드는 요소로 샛길 탐색에 있어서는 적합하지 않은 요인이다. 고려하는 weight는 차선 수에 대한 것인데 기존의 cost 계산방식과는 반대로 차선이 적을수록 cost가 낮아지므로 써 탐색 우선순위가 높아지게 된다. 기존 cost 계산에 사용된 weight들을 사용하면 차선 수에 대한 비중이 낮아져 원 경로에 가까워지는 현상을 보여준다.

기존 식(1)에서 회전cost의 비중을 없앤 다음 식을 사용한다.

TotalCost = (CurrentCost + NextCost)/2 (4)

이렇게 구해진 샛길경로는 타당성이 있는지 검토를 하게 되는데 샛길로써의 타당성이 떨어졌을 경우를 대비하여 다른 방식의 샛길 탐색을 한다.

두 번째 샛길 탐색은 첫 번째 샛길 탐색이 원 경로와 비교해서 원하는 목표수치를 달성하지 못했을 때 차선책으로 샛길을 탐색하는 방식으로 첫 번째 샛길 탐색과 비슷하지만 첫 번째 샛길 탐색에서 제외했던 회전cost를 포함시키는 방식이다. 하지만 회전cost를 그대로 사용하는 것이 아니라 회전 cost 의 역수를 취해서 이를 반영하기 때문에 원래 회전 cost가 가지는 의미인 직선에 가까운 경로가 아닌 회전이 많아지는 방향으로 회전cost가 쓰이게 된다.

기존 식(1)에서 회전cost의 비중을 줄이기 위해 다음과 같은 식을 사용한다.

TotalCost = (CurrentCost+NextCost)/2 + 30/TurnWeight (5)

2.2.2 도로 상황에 따른 샛길 정의 방법

위에서 찾아낸 두개의 샛길 경로를 원 최적경로와 비교하여 거리 증가 비를 판단한다.

거리 증가 비는 1.0일 경우 완벽히 일치하는 것이고 1.0 이상 일 경우 샛길이 원 경로보다 긴 경우이고 1.0 이하일 때는 샛길이 원 경로보다 짧은 경우 이다.

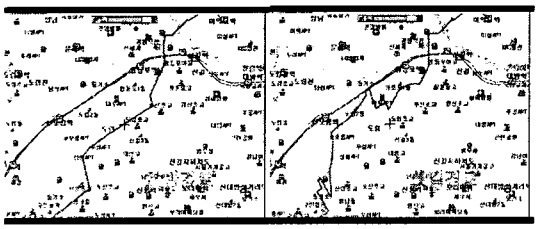
여기서 두 가지 경우가 생기는데 첫 번 째 는 고속도로 같은 구간에서의 샛길 탐색과 같이 긴 구간(5km 이상)의 샛길이 필요한 경우이다. 이 경우는 원 경로와의 거리증가비가 너무 크면 샛길의 가치가 떨어지므로 거리증가비가 적정 범위 안에 들어오는 샛길경로를 택한다. 현재 1.3의 거리 증가 비를 경계로 하고 있다. 1.3의 범위가 넘을 경우 그 샛길 탐색은 실패로 보고 차선의 샛길 탐색을 비교하여 그마저 1.3의 범위를 벗어날 경우 그냥 원 경로를 택한다.

두 번째 는 복잡한 시내에서 짧은 구간(5km이내)의 샛길을 구할 때 사용하는 방식이다. 이 경우에는 짧은 구간 내 에서 체증구간 탈출을 우선 목표로 하기 때문에 5km 내에서의 우회로 인한 거리증가는 무시하고 샛길과 원 경로와의 거리증가비가 클수록 좋은 것으로 보고 두 가지 방식으로 탐색된 샛길 경로를 비교하여 두 샛길경로중 거리증가비가 더 큰 경로를 선택한다.

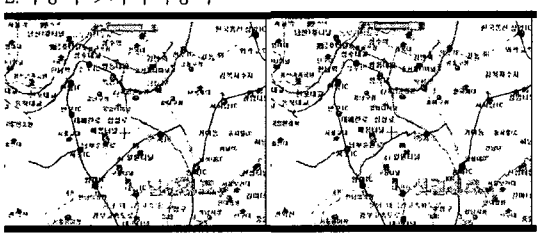
3. 결 론

지금까지 경로탐색을 위한 weighting 평가방법과 경로탐색 알고리즘, 샛길 탐색 알고리즘에 대해 살펴보았다. 다음은 샛길 탐색 알고리즘을 이용하여 실제 경로 탐색을 행한 결과이다. 먼저 샛길경로가 긴(5km이상)경우이다. 서울 시내를 중심으로 테스트하였다.

1. 여의도역->구로구청



2.사당역->가락시장역



각각의 거리증가비가 1.03 , 1.27로 설정 경계 값인 1.3 이하로 나타났으며 일반적인 탐색결과에서도 약 90% 확률로 1.3 경계 안으로 거리증가비가 나타났다.

다음은 샛길 경로가 5km이하인 구간에 대해 같은 구간 내에서 각 2, 3, 5km로 테스트한 결과이다.

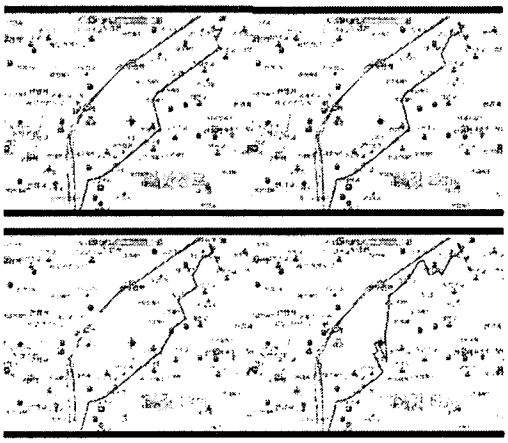
[1.강남구청 -> 가락시장]

거리증가비	첫번째 방식	두번째 방식
1km	1.00	1.00
2km	1.00	1.16
3km	0.99	0.99
5km	0.86	0.86

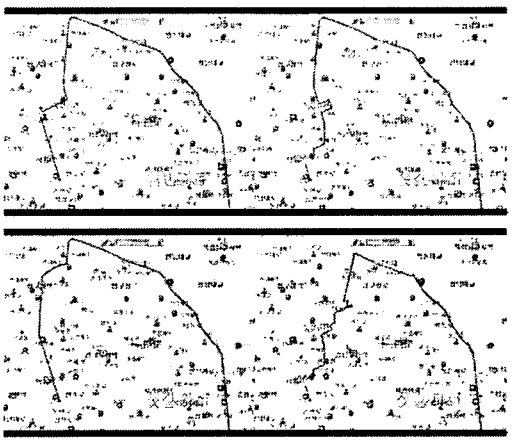
[2.여의도->구로구청]

거리증가비	첫번째 방식	두번째 방식
1km	1.00	1.00
2km	1.01	1.01
3km	0.99	0.99
5km	0.99	1.22

1.강남구청->가락시장



2.여의도->구로구청



현재 cost 계산 방식을 다양한 상황에 맞게 최적화 시킨다면 향후 경로 탐색서비스에서는 보다 나은 성능의 서비스를 선보일 수 있을 것으로 보인다. 현재 단순히 정보제공만으로 사용되고 있는 교통정보를 이용한다면 미리 교통 상황을 예측하여 자동적으로 샛길을 탐색하여 우회한 경로를 제공해 줄 수 있을 것이다.

[참 고 문 헌]

[1]Dijkstra . E. W.(1959), A note on two problems in connection with graphs, Numerische Mathematik I, pp 269-271.

[2]A* Pathfinding for Beginners, http://www.policyalmanac.org/games/aStarTutorial.htm

[3]Minchan Lee, Introduction to RP, Documents of Mobile Division/Development team 2 in Findigital Inc. October 22, 2002

[4]Bellman, R.(1958) On a routing problem, Quarterly applied Mathematics Vol. 16, pp 87-90.

[5]Yilin Zhao, "Vehicle Location and Navigation Systems", 1997

[6]M. A. Weiss, "Data Structures & Algorithm Analysis in Java", 1999