韓國OR學會誌
第8卷 第1號
1983年 4月

—硏究論文—

비행체의 經路最適化

(Path Optimization for Aircraft)

김 세 헌*
염 건*

Abstract

This paper shows a new efficient solution method of finding an optimal path for a cruise missile or aircraft to a target which has the maximal survivability and penetration effectiveness against sophisticated defenses and over varied terrain.

We first generate a grid structure over the terrain, to construct a network. Since our network usually has about 10,000 nodes, the conventional Dijkstra algorithm takes too much computational time in its searching process for a new permanent node. Our method utilizes the *Hashing technique* to reduce the computational time of the searching process.

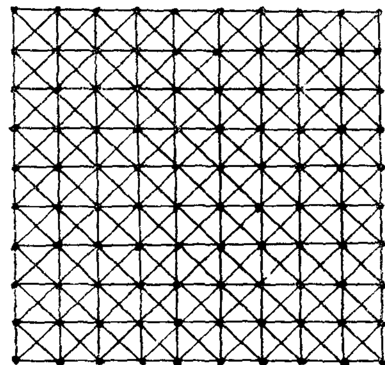
Extensive computational results are presented.

1. 序 論

最短經路問題(Shortest Path Problem)는 'Combinatorial' 최적화문제 中에서 가장 基本的이고 重要한 것일 뿐 아니라 상당수의 最適化問題를 이러한 最短經路問題로 바꾸어 해결할 수도 있다. 또 많은 複雜한 문제의 部分解로써도 종종 應用되고 있다.

本 論文에서는 이러한 最短經路問題를 미사일 공격計劃에 適用하는 경우, 즉 적군의 복잡하게 計劃된 방어망속에 있는 目標物을 我軍의 미사일 혹은 航空機로 공격하려고 할 때 과연 그 방어망 속을 어떤 經路로 뚫고 나아가야만 미사일의 殘存確率, 즉 침투효과를 크게 하여 궁극적으로 目的遂行力을 極大化 할 수 있겠는가 하는 問題를 다룰려고 한다. 이러한 문제를 經路最適化問題(Path Optimization Problem)라고 한다.

經路를 찾고자 하는 지역(terrain)을 連續的으로 다 考慮한다는 것은 불가능하기 때문에 格子構造(grid structure)를 생각할 수 있는데 이것은 對象地域을 가로, 세로 및 대각선으로 세분하여 그 교차점과 교차선을 가상의 마디(node)와 가지(arc)로 고려하여 最短經路 解法을 適用할 수 있는 基礎的인 構造를 意味한다. (〈그림 1〉參照)



〈그림 1〉 格子構造

그리고 여기에다 敵의 防禦網에 대한 情報를 토대로 敵의 포착기능 및 레이다, 미사일, 요격기 등의 위치를 좌표로써 표시하여 入力시켜 주면 그것으로부터 我

* 韓國科學技術院 經營科學科

軍미사일이 한 node에서 隣近 node까지 격추당하지 않고 성공적으로 비행할 확률을 계산할 수 있다. 이 확률값에 기초를 두고 미사일이 발사지점에서 목표물까지 성공적으로 도달할 확률이 제일 높은 경로를 구하는 問題를 다루고자 한다. 제 2 절에서는 이 문제가 〈그림 1〉과 같은 네트워크에서의 最短經路를 구하는 문제로 바뀌어 짐을 보여 준다.

이 문제의 효율적인 解法을 구하는 것은 단순히 컴퓨터의 효율적 이용이라는 점 이외에 매우 중요한 의미가 들어있다. 이 미사일(或은 항공기)이 地上에서 모든 情報(敵의 방어 System과 최적경로등)가 program되어 目標物을 向해 발사된다면, 그리고 더 以上의 조정(Control)이 없다면 最適經路를 地上에서 미리 계산하여 두게 되므로 最適經路를 찾는데 드는 時間은 큰 問題가 되지 않는다. 그러나 미사일에 컴퓨터를 내장시켜서 計劃된 經路를 따라 비행하는 도중 事前에 確認이 안되었던 새로운 情報를 얻게 되는 순간 새로운 最適經路를 다시 찾는다면 그때의 計算時間은 중요한 변수로 부각되게 된다. 새로운 經路를 빠른 시간내에 구하지 못한다면 계산도중 이 미사일은 敵의 공격을 받게 되기 때문이다.

실제의 經路最適化문제에 있어서 格子構造는 대개 약 10,000개 以上의 node를 갖게 되며 따라서 대형의 네트워크문제가 된다. 네트워크에서의 최단경로탐색해법중 효율적인 것으로는 Dijkstra[3]의 algorithm을 들 수 있다. 그러나 이 algorithm은 대형의 네트워크 문제에 있어서는 temporary node들 중에서 최소의 거리를 갖는 node를 구하는 과정에서 많은 계산시간을 소비한다. 이 論文에서는 이 계산시간의 소비를 줄이기 위하여 컴퓨터 科學技法中의 하나인 '해싱技法(hashing technique)'을 應用하여 最適解를 찾는데 걸리는 計算時間을 줄이고자 하였다. 3절에서는 해싱기법을 소개하겠다.

2. 經路 最適化 問題

敵軍의 방어망의 構造 및 위치, 그리고 지형에 관한 자료를 알면 我軍의 미사일이 한 node에서 隣近 node까지 피격되지 않고 비행할 확률을 구할 수 있다. 이 확률을 $P_{(i,j),(i',j')}$라 하자. 여기서 $|i-i'|\leq 1$, $|j-j'|\leq 1$이다. 이 확률은 적의 레이다 및 방어용 미사일 등의 성능과 지형에 기초를 두어 事前에 미리 계산된 데이타이다. 이 데이타에 기초를 두어서 발사지점에서 목표물까지 피격되지 않을 확률(殘存確率)을 극대화하는 經路를 구하려는 것이 우리의 問題이다.

(i_s, j_s)를 발사지점의 node, (i_t, j_t)를 목표지점의 node라고 하자. 그러면 발사지점에서 목표지점까지의 經路, r는 다음과 같이 나타낼 수 있다.

$$r=\{(i_0, j_0), (i_1, j_1), \cdots, (i_K, j_K)\},$$
$$\text{단 } (i_0, j_0)=(i_s, j_s),\ (i_K, j_K)=(i_t, j_t),$$
$$|i_k-i_{k+1}|\leq 1,\ |j_k-j_{k+1}|\leq 1$$
$$\text{for all } k=0, \cdots, K-1$$

그러면 경로 r의 殘存確率, P_r는

$$P_r=\prod_{k=0}^{K-1} P_{(i_k,j_k),(i_{k+1},j_{k+1})}.$$

여기서 D_r를 다음과 같이 정의하자.

$$\begin{aligned} D_r &= -\ln P_r \qquad (1)\\ &= -\ln\left[\prod_{k=0}^{K-1} P_{(i_k,j_k),(i_{k+1},j_{k+1})}\right]\\ &= \left[\sum_{k=0}^{K-1} -\ln P_{(i_k,j_k),(i_{k+1},j_{k+1})}\right]\\ &= \sum_{k=0}^{K-1} D_{(i_k,j_k),(i_{k+1},j_{k+1})}, \end{aligned}$$

$$\text{단 } D_{(i_k,j_k),(i_{k+1},j_{k+1})} \equiv -\ln P_{(i_k,j_k),(i_{k+1},j_{k+1})}.$$

P_r를 최대화하는 經路는 D_r를 최소화하는 經路가 되므로 이 문제는 $D_{(i_k,j_k),(i_{k+1},j_{k+1})}$를 node$(i_k, j_k)$와 (i_{k+1}, j_{k+1})사이의 '거리'라고 생각하면, 소위 '最短經路問題'가 된다. 여기서는 D를 '위험도(Danger)'라고 부르겠다. 이와같은 이름의 타당함을 보이기 위하여 다음의 성질을 들 수 있다.

1) 잔존확률이 크면 D는 작아진다.

2) 적의 위협이 없으면($P=1$이면), $D=0$이다.

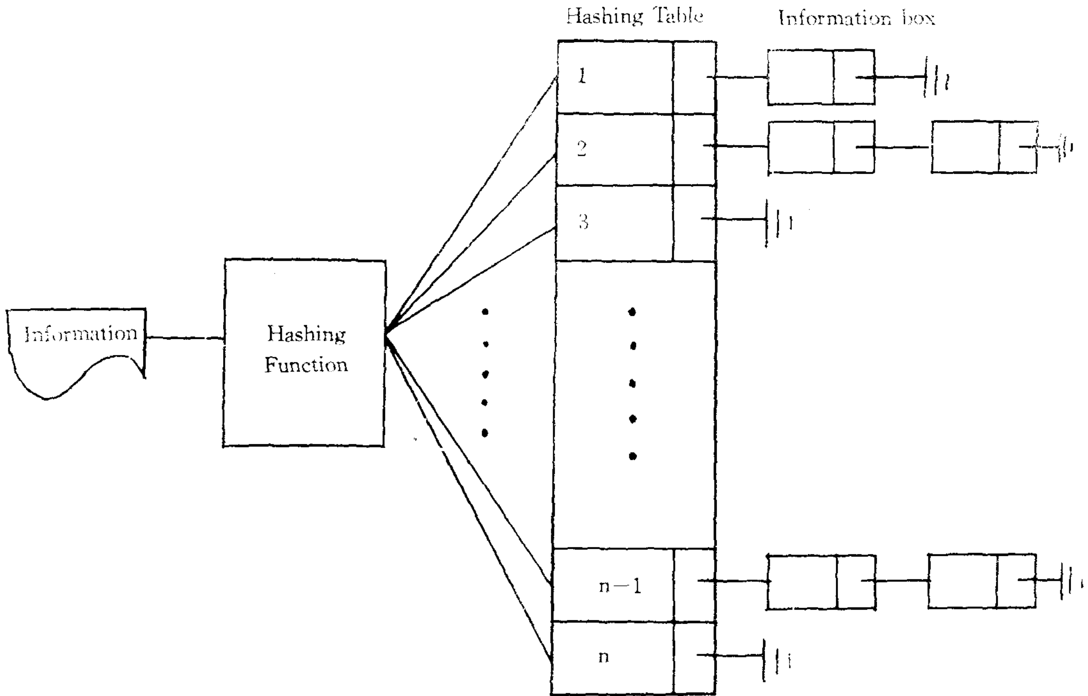
3) 적의 방어가 완벽하면 ($P=0$이면), $D=\infty$이다.

특히, D는 비음수(nonnegative)이므로 Dijkstra agorithm을 적용해서 이 문제의 解를 구할 수 있다.

여기서는 Dijkstra algorithm을 기본적인 틀로하고 이 algorithm中 계산시간이 많이 걸리는 탐색과정(search process)을 보다 효율적으로 하기 위하여 Denardo와 Fox의 'Bucket 槪念'[2]과 컴퓨터 科學技法中의 하나인 해싱技法을 도입하였다.

3. 해싱技法

이것은 우편함체계를 通해서 편하게 우편물을 받아볼 수 있는 것을 예로 들 수 있다. 다시말하면 고유의 우편함을 지정해 주는 것이 바로 해싱 함수이다. 그리고 이 해싱함수에 의해서 찾아진 우편함들이 Denardo와 Fox의 bucket과 같은 槪念이다. 또 우편물은 바로 이 bucket속에 들어갈 node인데, 여기서는 情報상자



〈그림 2〉 해싱技法

(*information* box)라고 하여 이 node에 관련된 모든 情報를 담고 있을 뿐만 아니라 어느 bucket, 或은 어느 다른 情報상자에 연결되어 있는 가를 表示하는 역할도 한다. 이러한 관계를 〈그림 2〉가 보여준다.

우리의 네트워크문제에서는 '거리'를 작은 구간들로 잘라서 각 구간들을 하나의 bucket으로 잡는다. Dijkstra algorithm의 모든 temporary node들은 starting node에서부터의 거리가 속하게 되는 bucket에 저장된다. 그러면 모든 temporary node들 중에서 최소의 거리를 갖는 node 구하는 search process는 간단히 이 bucket들을 앞에서 부터 check하며 처음의 nonempty bucket을 찾아내면 된다. (이 bucket 안에 여러 *node*가 있는 경우에는 이들 중에서 제일 작은 node를 찾으면 된다.)

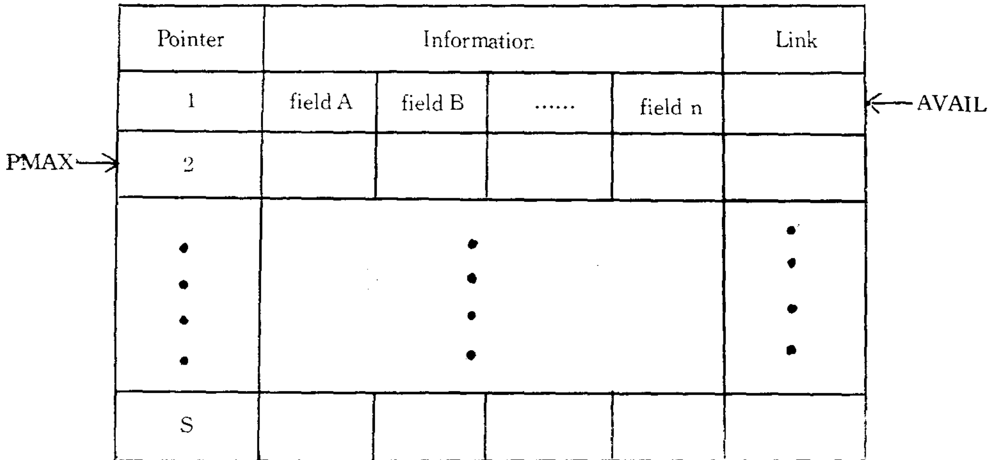
그 다음 *iteration*에서 *nonempty* bucket을 찾을 때는 그 전의 iteration에서 찾은 bucket에서 부터 시작하면 된다. 따라서 nonempty bucket을 찾는 pointer는 한 방향으로만 움직이게 되어 불필요한 과정의 반복을 하지 않아도 된다.

bucket의 크기 및 갯수는 미리 지정해 주게되므로 두 node가 비슷한 거리를 갖게 되면 이 두 node는 같은 bucket에 들어가게 될 수 있다. 따라서 각 bucket은 여러개의 기억장소를 배당받아야 하는데 너무 많이 받으면 기억장소가 너무 많이 필요하게 되고 너무적게 받으면 어떤 bucket에는 기억장소가 모자라게 되는 수가 있다. 이를 해결하기 위하여 어디엔가 한꺼번에 최소로 필요한 만큼만 storage pool을 만들어 놓고 필요한 때마다 가져다 쓰고 더 이상 필요가 없게 되면 도로떼어 storage pool에 넣어두면 記憶場所를 상당히 줄일 수 있게 된다. [1] 다만 이때 필요한 것은 使用되고 있는 情報상자가 어느 bucket, 或은 어느 情報상자에 연결되어 있는가를 나타내 주는 연결체계(*linkage* system)이다. 〈그림 3〉은 이 storage pool을 보여 주고 있다.

4. Algorithm

먼저 Denardo와 Fox의 생각처럼 有限한 값을 갖는 node만 고려의 대상으로 하였다. 즉, 無限한 값을 갖는 node(現在 고려中인 node에서 到達不可能한 node)들은 search process에서 제외된다. 이 algorithm에서는 Denardo와 Fox의 論文에서처럼 bucket의 幅을 均等하게 나누지 않고, node값이 밀집되어 있는 곳은 세분하고 그렇지 않은 곳은 크게 하나의 bucket으로 하는 方法을 使用했다.

Denardo와 Fox의 Bucket Method는 출발점에서부



〈그림 3〉 **Storage pool**

여기서 S: Storage Pool에 마련된 情報상자의 최대갯수
Pointer: 정보상자의 번호
情報 : node (i, j)까지의 거리 또 그와 關聯된 資料
Link: 情報상자가 연결되는 곳의 번호 或은 위치
PMAX: 現在 使用中인 情報상자의 最大 숫자
AVAIL: 現在 未使用인 情報상자의 번호로, 情報상자가 必要할 때 우선적으로 使用될 番號를 말한다.

터 모든 arc를 계속 이어놓고 均等한 幅으로 bucket을 만들어 가는 것인데 이렇게 하면 그 길이가 殘存確率에 로그함수를 취한 값이어서 거리가 멀어질수록 殘存確率의 구간은 점차로 좁아지게 된다. 즉, 殘存確率이 1에서 0으로 갈수록 단위 bucket內에 들어오게 될 node의 수는 점차 적어질 것이므로 그 部分을 계속해서 細密하게 區分할 필요는 없게된다.

따라서 本 algorithm에서는 node가 密集되어 있는 部分은 좁게 그리고 그렇지 않은 部分은 넓게 bucket의 幅을 잡아서 단위 bucket內에 떨어질 node의 수를 최소로 하였다. 그래서 먼저 발사지점에서 가장 가까운 node까지의 殘存確率(SP_0)과 殘存確率이 0.5[1] 되는 部分까지를 1999개의 bucket으로, 그리고 나머지를 한 개의 bucket으로 하여 모두 2000개의 bucket으로 하였다.

이 algorithm에서 해싱함수로는 CYBER 174—16컴퓨터의 내장함수(intrinsic function)인 IFIX function을 사용했다. 이것은 그 수를 넘지 않은 最大 정수를 얻기 위하여 使用되었는데, 例를들면 bucket의 幅이 w이고 node (i, j)까지의 tentative label이 D_j라면

〈註 1〉 0.5라는 값은 結果的이기는 하나 10개의 例題를 풀어본 결과, 최종 遂行確率이 항상 0.5(50%) 이상으로 나타난 것으로 부터 결정되었다. 그러나 상황이 바뀌는 경우 이 값은 0.0~1.0사이의 어느 값으로든 대체할 수 있으며, 이것은 큰 問題가 되지 않는다.

$$1+\text{IFIX}\,[\{\exp(-(D_{ij}-SP_0))\}/w]$$

가 node(i, j)의 情報가 記憶되는 bucket의 번호를 나타내게 된다. (위에서 $\exp\{-(D_{ij}-SP_0)\}$는 2절의 식(1)에서 볼 수 있는 것처럼 D_r가 주어지면 $P_r=\exp(-D_r)$이므로 길이를 다시 잔존확률로 환산한 값이 된다.)

이와같이 bucket을 결정하고 난 다음, 同一한 bucket 속에서 tentative label의 크기 순으로 (작은 것을 앞쪽, 큰 것을 뒤쪽으로) 여러 node의 情報를 기억해 둔다. 그리고 가장 최소값을 갖는 node를 찾는 경우(search process) 우선 情報를 가지고 있는 최초의 bucket을 찾은 다음, 그 中 제일 첫번째의 情報상자를 택하면 된다. 이때 2,000개의 bucket 수가 많은 것으로 여겨질른지 모르나 algorithm이 進行되면 될수록 거리는 누적적으로 커지는 것이므로 탐색과정에서 단지 現在 찾고 있는 bucket과 그 다음의 몇몇 bucket만을 찾아보면 된다. 이 algorithm을 요약하면 다음과 같다.

Hashing技法을 쓴 Dijkstra algorithm

단계 0 (시작)

$u_{st}=0$로 한다. (node(s, t)는 발사지점)

$u_{ij}=D_{st,ij}$로 한다. ($(i, j)\neq(s, t)$이고 (s, t)의 둘레에 있는 점이다.)

$P=\{(s,t)\}$, $T=\{(i,j)|i=1,\cdots,M\ j=1,\cdots,N$, 그리고 $(i,j)\neq(s,t)\}$로 한다.

$SP_0(=\exp(-D_{st,xy}))$를 구한다. (여기서 node(x,y)는 발사지점에서 가장 가까운 node이다.)

단계 1 (行別로 최소값을 갖는 node를 찾음)

$u_{l^jj}=\min_i \{u_{ij}\}$인 node$(l^jj)$를 찾는다. $(j=1,\cdots,N)$

단계 2 (Hashing table의 bucket을 결정함)

有限한 값을 갖는 (l^j,j)에 대해서만 다음과 같이

$1+\mathrm{IFIX}[\{\exp(-(u_{l^jj}-SP_0))\}/w]$

하여 bucket을 결정한 후 情報상자를 가져다 크기순으로 記憶한다.

단계 3 (Permanent label을 가질 node를 찾음)

정보상자를 달고 있는 첫번째 bucket을 찾은 후 그 중 첫번째 node(l^k,k)를 택한다.

$T=T-\{(l^k,k)\}$, $P=P+\{(l^k,k)\}$로 한다. 그리고 node (l^k,k)의 정보상자를 떼어낸다.

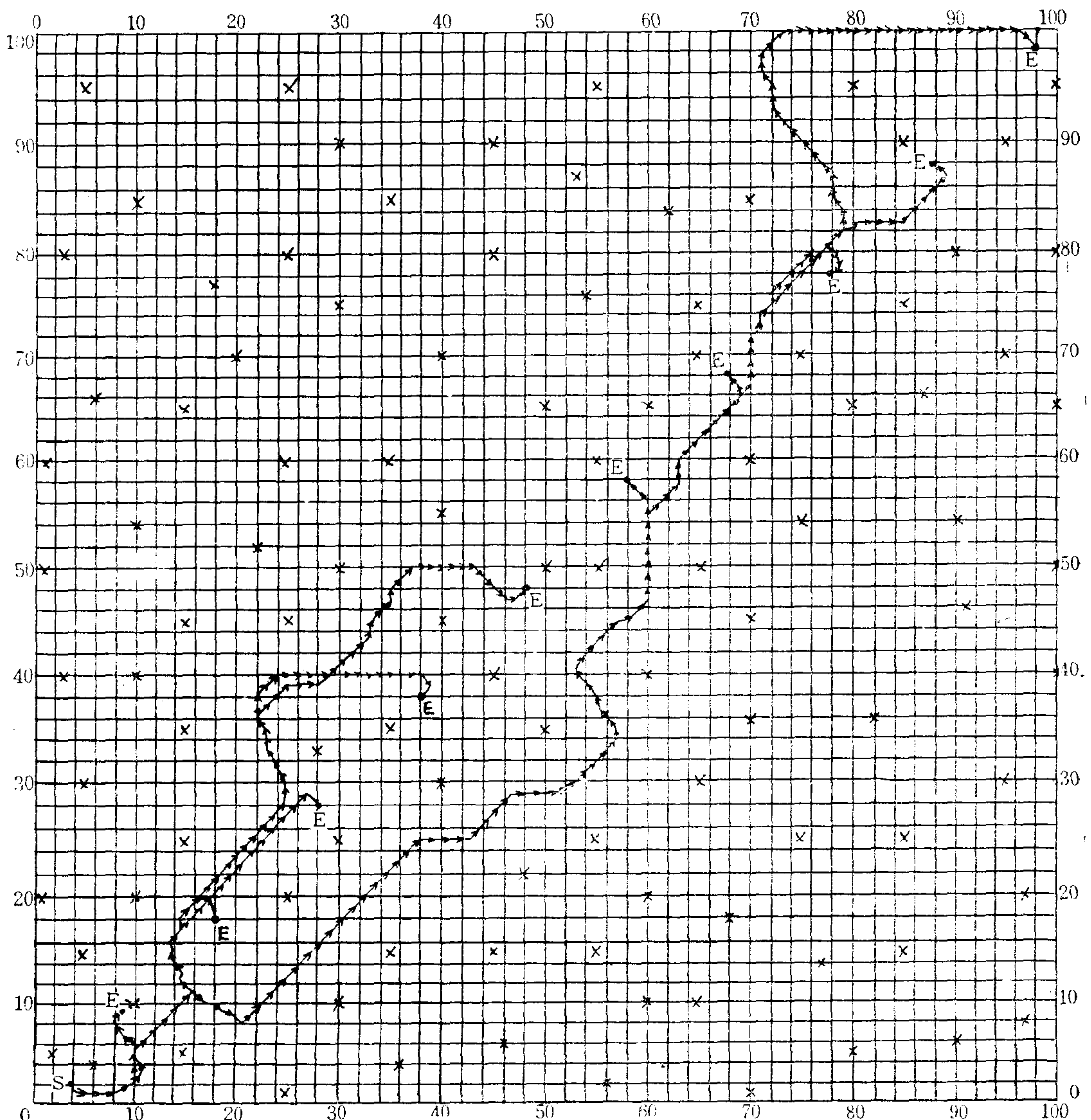
만약 $(l^k,k)=(u,v)$이면 計算은 끝낸다.

(여기서 node (u,v)는 목표물이다.)

단계 4 (Tentative label의 修訂)

$u_{ij}=\min\{u_{kj}, u_{l^kk}+D_{l^kk,ij}\}$로 한다.

(이때 $(i,j)\in T$이면서 (l^k,k)주변에 있는 것만 고려한다.)



〈그림 4〉 10가지 例題의 最適經路들

단계 5 (行別로 최소인 node 값의 修訂)

行 $k-1$, k, $k+1$인 경우만 $u_{lij}=\min_{\substack{i \\ (i,j)\in T}} \{u_{ij}\}$로 한다.

이때 새롭게 치환되는 경우는 먼저번의 값을 담고 있던 정보상자를 떼어낸다. 그리고 단계 2로 간다. 그렇지 않으면 바로 단계 2로 간다.

5. 結果 및 分析

5.1. 人力資料 및 出力內容

기존의 Dijkstra algorithm과 해싱技法을 쓴 Dijkstra algorithm의 效率을 비교하기 위하여 格子構造 크기 10×10, 20×20, …, 100×100인 10개의 例題를 만들어 Test 해 보았다.

여기서 入力資料로 필요한 것은 格子構造의 行과 列의 수와 敵防禦物의 位置와 발사지점과 目標物의 位置를 좌표로써 나타낸 것이다. 이 資料로 부터 格子構造上의 모든 arc의 길이를 구할 수 있고 이것으로 부터 원하는 最短經路를 구하게 된다.

出力內容은 최적수행확률과 최단경로를 나타내는 control로 되어 있다. 여기서 最適 control이라 함은 目標物에서 발사지점까지의 순서로 각각 그 node로 들어오는 arc의 方向을 나타내는 숫자를 말한다. (즉, 1=↑, 2=↗, 3=→, 4=↘, 5=↓, 6=↙7=←, 8=↖이다)

〈그림 4〉는 10개의 例題問題와 그 결과 얻어진 최단경로를 한꺼번에 나타낸 것이다. S는 발사지점(我軍의 미사일 기지), E는 目標物, X는 敵의 防禦物을 나타낸다. 편의상 格子 2개에 한줄로만 表示하고 대각선 arc는 생략했다.

5.2. Algorithm의 比較

〈表 1〉은 conventional Dijkstra algorithm과 해싱技法을 쓰는 Dijkstra algorithm을 使用하여 10가지 예제를 풀었을 때의 계산시간을 비교해 본 것이다. 비교를 정확히 하기 위하여 컴퓨터 계산시간을 3部分으로 나누었다. 즉,

- Search Time: Dijkstra algorithm의 단계 1, 즉 최소값을 갖는 node를 찾는데 걸리는 時間.
- Update Time: Dikstra algorithm의 단계 2, 즉 tentative label들을 修訂하는데 걸리는 時間
- Generation Time: 모든 arc길이를 data file에서 읽어 들이는데 걸리는 時間.

이다.

表에서 볼 수 있는 것처럼 conventional Dijkstra algorithm보다 해싱技法을 쓴 Dijkstra algorithm이 상당히 效率的인 것을 알 수 있다. 그리고 문제의 규모가 커질수록 점차 좋은 效率을 보이는 것도 알 수 있다. 〈그림 5〉는 이러한 效率을 잘 나타내고 있다.

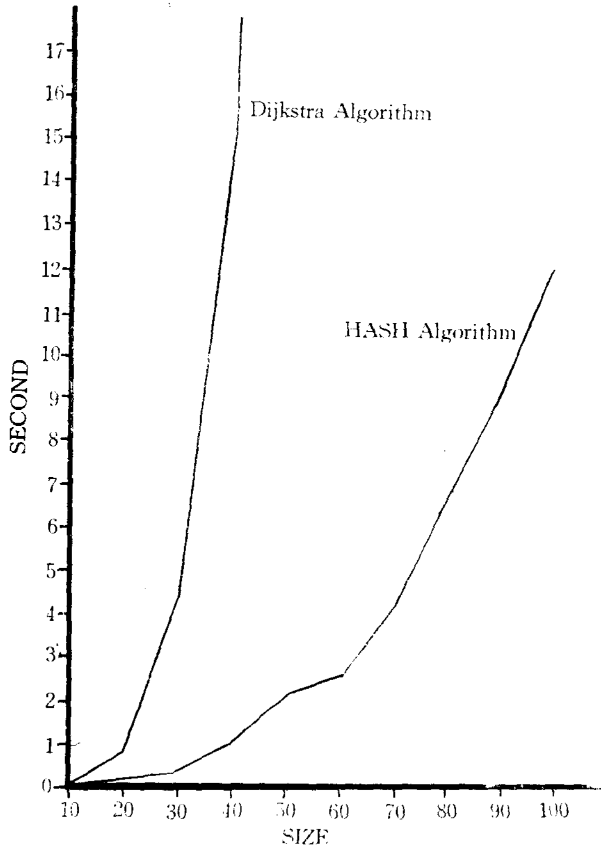
〈表 1〉 컴퓨터 計算時間의 比較

격자의 수 \ CPU, second		Total time	Search time	Update time	Generation time
10×10	D	0.508	0.099	0.018	0.278
	H	0.477	0.077	0.011	0.283
20×20	D	2.239	0.899	0.061	1.115
	H	1.566	0.217	0.044	1.123
30×30	D	7.298	4.296	0.136	2.582
	H	3.480	0.487	0.118	2.549
40×40	D	20.124	14.854	0.256	4.521
	H	6.480	1.105	0.233	4.525
50×50	D	51.660	43.305	0.448	7.019
	H	10.983	2.191	0.428	7.150
60×60	D	102.473	86.624	0.631	10.306
	H	14.599	2.511	0.446	10.319
70×70	D	168.023	152.835	0.723	13.993
	H	20.699	4.135	0.668	13.922
80×80	D	293.254	272.601	1.027	18.203
	H	27.919	6.225	0.955	18.082
90×90	D	………	………	………	………
	H	36.879	8.884	1.341	23.150
100×100	D	………	………	………	………
	H	46.208	11.710	1.698	28.550

※ 參考 : …는 더이상 고려할 必要가 없어 省略함.
: CYBER 174—16컴퓨터 使用
D : conventional Dijkstra algorithm
H : 해싱 技法을 쓴 Dijkstra algorithm

6. 結　論

最短經路를 구하려고 하는 地域을 보다 세밀한 格子로 나누면 나눌수록 더욱 정밀한 解를 찾을 수 있을 것은 分明하다. 그러나, 이것이 미사일 或은 항공기의 비행경로를 나타내는 本 硏究의 경우에는 그들이 經路를 바꿀 수 있는 최소반경이 格子한 단위의 크기가 되어야 하고 그것보다 작은 格子 크기는 실행이 不可能



〈그림 5〉 **Search time의 比較**

하므로 사실상 의미가 없게 된다. 뿐만 아니라 세분하면 할수록 計算時間은 기하급수적으로 증가하게 된다.

이러한 문제는 사실상 별로 필요치 않은 部分을 細密한 格子로 나누고 있는데서 기인한다. 그러나 미사일 방어망을 보면 전략상 중요한 지점은 상당히 세밀한 經路가 필요할 만큼 복잡하게 계획되어 있는 것은 사실이다. 그러므로 格子크기를 변동적으로 잡아, 복잡하게 계획된 방어망 속에서는 세밀하게' 별로 그렇지 않은 곳에서는 넓게 格子 크기를 잡아 준다면 해를 얻기 위한 계산시간을 상당히 줄일 수 있으리라 생각된다. 이와 같이 한다면 더욱 큰 問題도 다룰 수 있겠고 해싱技法이 그 連結體系가 복잡한 것이기 때문에 문제의 크기가 커질수록 效果는 더욱 커지게 될 것이다. 이것은 앞으로 계속 硏究할 課題로 남는다.

格子構造는 특별한 方向이 없는 네트워크이므로 위와 오른쪽 방향을 'forward'로 하고 아래와 왼쪽 방향을 'backward'로 생각하며 forward DP과 backward DP을 번갈아 가며 적용하면서 두 解가 같아졌을때 최적解를 求하는 방법인 Multi-pass Dynamic Programming Algorithm을 사용할 수도 있는데 이 경우 크기가 작은 문제에서는 네트워크最適化 理論이 效率的이라고 알려져 있다. [6] 그리고 대형문제의 경우도 해싱技法을 쓴 Dijkstra algorithm이 해싱技法을 사용하여 search time을 상당히 줄이고 있기 때문에 위의 DP algorithm보다 效率的일 것이라고 생각된다. [6]

한가지 더 언급할 것은 각 bucket에 들어오는 node 수의 분포를 분석한다면 bucket의 幅을 변동적으로 잡을 수 있을 것이고 따라서 일단 解를 구한다음 새로운 情報에 의하여 데이타가 약간 수정됐을 경우 위에서 구한 변동적인 bucket 幅을 사용한다면(즉 warm start의 경우) 탐색시간(search time)은 더욱 줄어들 것이다. 이것은 앞으로 계속 硏究할 課題이다.

7. References

1. C.J. Date, *An Introduction to Data-Base Systems*, 2nd Edition, pp. 27~49, Addition-Wesley Publishing Company, 1977.
2. E.V. Denardo and B.L. Fox, "Shortest Route Methods; 1. Reaching, Pruning, and Buckets,"

Opns, Res. 27, 161~186(1979).

3. E.V. Dijkstra, "A Note on Two Problems in Connecting with Graphs," *Numerische Mathematik 1,* 269~271 (1959).
4. S.E. Dreyfus, "An Appraisal of Some Shortest-Path Algorithms," *Opns. Res. 17,* 395~412 (1969).
5. B.L. Fox, "Data Structures and Computer Science Technique in Operations Rearch," *Opns, Res. 26,* 686~717(1978).
6. S. Kim, "Single-Path Dynamic Programming Algorithm," *Technical Memo in Systems Control Technology,* Inc. (1982).
7. R.E. Larson and J.L. Casti, *Principles of Dynamic Programming, Part I,* Marcel Dekker, Inc, 1978.
8. E.L. Lawler, *Combinatorial Optimization: Networks and Matroids,* pp. 59~108, Holt, Rinehart and Winston, 1976.