다중 동적구간 대기행렬을 이용한 최단경로탐색 알고리즘

김 태 진[†] · 한 민 홍[††]

요 약

본 논문에서는 네트워크 그래프의 최단경로탐색 문제에서 후보노드집합(Candidate Node Set)의 특성을 분석하고, 새로운 다중 동적구간 대기행렬(Multiple Dynamic-Range Queue : MDRQ)구조를 제안하여 후보노드집합을 관리함으로써 최단경로탐색 시간을 단축시킨다. 다중 동적구간 대기행렬 알고리즘은 연산이 진행되면서 후보노드집합의 분포가 일정한 상수분포 형태를 유지하게 되는 특성과 후보노드집합의 크기가 급격하게 변하는 특성을 고려하여, 후보노드집합을 관리하는 대기행렬의 범위와 크기를 유연하게 변동시키는 최단경로탐색 알고리즘이다. 본 알고리즘은 꼬리표개선(Label-Correcting) 알고리즘 계열에 속하면서도 후보노드의 재진입을 상당히 줄일 수 있어 최단경로탐색 시간을 현저히 감소 시킬 수 있다. 실험을 통하여 본 알고리즘은 재진입이 자주 발생하지 않는 그래프에서 다른 꼬리표개선 알고리즘들과 유사하거나 우수했으며, 재진입이 자주 발생하는 그래프에서는 다른 꼬리표개선 알고리즘들보다 월등히 우수했고, 꼬리표설정(Label-Setting) 알고리즘 계열보다 20% 성능이 향상되는 결과를 보였다.

Shortest Path-Finding Algorithm using Multiple Dynamic-Range Queue (MDRQ)

Tae-Jin Kim[†] · Min-Hong Han[††]

ABSTRACT

We analyze the property of candidate node set in the network graph, and propose an algorithm to decrease shortest path-finding computation time by using multiple dynamic-range queue (MDRQ) structure. This MDRQ structure is newly created for effective management of the candidate node set. The MDRQ algorithm is the shortest path-finding algorithm that varies range and size of queue to be used in managing candidate node set, in considering the properties that distribution of candidate node set is constant and size of candidate node set rapidly change. This algorithm belongs to label-correcting algorithm class. Nevertheless, because re-entering of candidate node can be decreased, the shortest path-finding computation time is noticeably decreased. Through the experiment, the MDRQ algorithm is same or superior to the other label-correcting algorithms in the graph which re-entering of candidate node didn't frequently happened. Moreover the MDRQ algorithm is superior to the other label-correcting algorithms and is about 20 percent superior to the other label-setting algorithms in the graph which re-entering of candidate node frequently happened.

키워드 : SPT, MORQ, Shortest Path, ITS

1. 서 론

최단경로탐색 문제는 시점노드에서 종점노드까지 거리가 가장 작은 경로를 찾아내는 문제로, 수송망과 통신망에서 발생하는 문제들의 근본적인 인식되어 오고 있다. 특히 ITS (Intelligent Transport Systems)분야에서 차량의 시점노드에서 종점노드까지의 경로를 실시간으로 알려주려는 시스템의 필요성으로 인하여, 본 문제의 중요성은 시간이 갈수록 증가하고 있다.

최단경로탐색 문제는 해결방법, 목적, 제약조건에 따라서 많은 알고리즘이 제시되어 왔고, 이러한 최단경로탐색 문제를 Deo와 Pang은 체계적인 방법으로 분류하였다[1]. 1996년에 Cherkassky, Goldberg, Radzik이 알고리즘에 따른 분류와 요약, 일정한 실험환경을 가지고 성능분석을 하였다[2].

최단경로탐색 문제를 해결방법에 따라 분류하면 꼬리표설정 알고리즘(Label-Setting) 계열과 꼬리표개선(Label-Correcting) 알고리즘 계열로 나눌 수 있다. Dijkstra는 꼬리표설정 알고리즘 계열의 모태가 되는 알고리즘을 제안하였다[3]. 꼬리표설정 알고리즘 계열은 탐색 완료된 집합에 속하는 노드에 연결된 가장 가까운 후보노드를 선택하고, 그 선택된 후보노드를 탐색 완료된 집합에 포함시킨다. 이렇게 새로 포함된 후보노드는 시점노드에서부터의 거리가 최단임을 증명할 수 있다. 꼬리표설정 알고리즘 계열의 성능을

† 준 회 원 : ㈜모토로라 코리아 테크놀러지 센터-CDMA 소프트웨어팀 선임연구원
†† 정 회 원 : 고려대학교 산업공학과 교수
논문접수 : 2000년 11월 27일, 심사완료 : 2001년 5월 2일

결정짓는 요인은 후보노드집합 중에서 가장 가까운 노드를 선택하는 방법이다. 새로운 관점으로 후보노드 선정 시에 후보노드집합에서 최단후보노드를 선택하지 않고 일정한 규칙에 따라 후보노드를 선택하여 탐색 완료된 집합에 포함시킨 후, 탐색 완료된 집합 전체의 최단조건을 검사하여 조건에 위배되는 노드들을 다시 후보노드집합에 포함시키는 방법이 제안되었다[4, 5]. 이 꼬리표개선 알고리즘 계열 방법에서 성능을 결정짓는 요소는 탐색 완료된 집합에 속하는 노드가 후보노드집합에 재진입하는 회수를 최대한 줄이는 것이다.

최단경로탐색 문제를 목적에 따라 분류하면 시점노드와 종점노드가 각 하나인 경우, 동일한 시점노드에서 모든 노드까지 경로를 탐색하는 경우, 모든 노드사이의 경로를 탐색하는 경우로 나눌 수 있다. 모든 노드사이의 경로를 구하는 문제는 Floyd-Warshall 알고리즘을 기반으로 경로를 탐색하는 알고리즘들이 주를 이룬다[6]. 한편 시점노드와 종점노드가 각 하나인 경우와 동일한 시점노드에서 모드 노드까지 경로를 탐색하는 알고리즘은 동일한 알고리즘을 사용하여 탐색한다. 두 가지 알고리즘의 차이는 알고리즘의 종료조건이 종점노드가 하나인 경우는 종점노드까지 최단거리임을 증명할 수 있으면 알고리즘을 종료할 수 있다는 것이다. 꼬리표설정 알고리즘 계열에서 탐색 완료된 집합에 속하는 노드는 시점노드에서의 거리가 이미 최단거리임을 증명할 수 있으므로, 종점노드가 탐색 완료된 집합에 속하는 즉시 알고리즘 탐색을 종료할 수 있다. 반면에 동일한 시점에서 모든 노드까지 경로를 탐색해야 할 경우는 모든 노드가 탐색 완료된 집합에 속할 때까지 알고리즘을 종료하지 않는다. 꼬리표개선 알고리즘 계열은 종점노드가 하나인 경우에 일반적으로 알고리즘을 종료하는 것이 불가능하므로, 모든 노드까지 경로를 탐색해야 하는 경우와 알고리즘의 차이가 나지 않는다.

최단경로탐색 문제에서 종점노드가 하나인 문제는 시점노드와 종점노드의 거리가 매우 먼 경우 탐색시간이 많이 걸리는 문제에 초점을 맞추고 있다. 즉 시점노드와 종점노드가 가까운 경우는 알고리즘 탐색시간이 매우 짧아 성능의 차이가 나지 않는다. 시점노드와 종점노드가 매우 먼 경우, 알고리즘은 종점노드가 하나인 경우와 모든 노드까지 경로를 탐색하는 문제가 동일한 과정을 수행하게 된다.

꼬리표설정 알고리즘은 후보노드집합에서 항상 누적거리가 최소인 노드를 찾아낼 수 있어야 하며, 후보노드집합에 존재하는 노드의 누적거리는 탐색이 진행됨에 따라 값이 계속 변하는 특성을 가진다. 꼬리표설정 알고리즘은 후보노드집합을 관리하는 방법에 따라 성능이 달라지며, 알고리즘의 성능은 시간한계(Time Bound)와 그래프의 복잡도에 따른 실험결과를 토대로 알고리즘의 성능을 판단한다. 일반적으로 꼬리표설정 알고리즘 계열은 시간한계 성능이 매우 우수한 반면, 후보노드집합을 관리하는 부분에서 연산을 상대적으로 많이 하여 실험조건에 따라 성능이 매우 나빠지는 경우가 많다.

꼬리표설정 알고리즘은 후보노드집합 관리를 위하여 기본구조로 힙(Heap)을 사용하는 방법과 버킷(Bucket)을 사용하는 방법으로 나뉘어진다. 힙은 노드 각각에 누적거리 값을 가지는 배열구조로 후보노드집합에서 최소후보노드를 한번에 찾아낼 수 있지만, 그에 따른 후속 후보노드집합의 정리에 많은 시간을 소비할 수 있다. 힙을 사용한 알고리즘은 링크거리 값이 매우 크거나 편차가 있고, 후보노드집합 변경이 적은 그래프에서 좋은 성능을 나타내는 알고리즘이다. 현재 Fibonacci힙을 사용한 알고리즘이 $O(m + n\log n$: m(링크의 수), n(노드의 수))의 시간한계를 가져 가장 우수한 알고리즘으로 알려져 있으나, 실제로 실험환경에서는 Fibonacci 힙 알고리즘의 복잡성으로 인하여 Binary 힙이나 d-Heap을 사용했을 경우와 큰 차이가 나지 않거나 오히려 탐색시간이 더 많이 걸린다[7]. 한편 버킷을 기본구조로 하는 알고리즘은 Dial이 $O(m + nC$: m(링크의 수), n(노드의 수), C(모든 링크중 최대거리))의 시간한계를 가지는 알고리즘을 처음 제안하면서 시작되었다[8]. 버킷을 사용한 알고리즘은 힙을 기본구조로 하는 알고리즘과 반대로 누적거리 값 각각에 노드번호를 가지는 배열구조로 되어 있다. 버킷을 사용한 알고리즘은 그래프 복잡성에 상관없이 매우 우수한 성능을 나타내는 반면, 링크거리에 따라 성능이 크게 차이가 나는 특성을 가지고 있다. 링크거리 변화에 따른 성능 저하를 만회하려고, Double Bucket 알고리즘이 Cherkassky, Goldberg, Radzik에 의하여 제안되었다. 이 알고리즘은 대체적으로 많은 실험환경에서 비교적 안정된 연산결과를 나타내었다[2]. 그러나 이 알고리즘은 링크의 거리값이 소수일 경우 처리가 불가능하며, 후보노드집합 수가 비교적 적은 알고리즘에서 상대적으로 시간이 많이 걸리는 약점을 가지고 있다. 힙과 버킷 두 가지 장점만을 사용하는 Cherkassky와 Goldberg에 의해 제시되었다. 이 알고리즘은 후보노드집합에 존재하는 후보노드의 수가 적을 경우에는 힙으로 동작하고, 많은 경우에는 버킷으로 동작하여, 두 알고리즘의 최악의 경우를 개선하는 데 중점을 두었다[9]. 그러나 실제로 두 가지의 후보노드집합을 관리해야 하는 처리시간 부담을 가지고 있어, 일반적인 경우에는 다른 알고리즘 비하여 성능이 나빠지는 특성을 나타냈다[9].

꼬리표개선 알고리즘 계열의 성능을 결정하는 요소는 후보노드집합에서 탐색완료 시까지 처리하는 노드 수와 후보노드집합을 관리 소요시간의 합으로 결정되어진다. 꼬리표설정 알고리즘 계열은 노드탐색 수를 최소화하면서 후보노드관리에 많은 시간을 소요하는 반면, 꼬리표개선 알고리즘 계열은 후보노드집합 관리시간을 최소화하여 알고리즘의 성능을 향상시키려 한다. 꼬리표개선 알고리즘은 후보노드집합에 새 후보노드를 추가하여 정렬하는 방법에 따라 알고리즘 성능이 크게 차이 난다. 그러므로, 탐색 완료된 집합에 속하는 노드들이 다시 후보노드집합에 재진입하지 않도록 후보노드집합을 관리하면서 재배열에 필요한 정렬시간 또한 최소화 하여야 한다.

Bellman과 Ford는 후보노드집합을 FIFO대기행렬로 관리

하는 꼬리표개선 알고리즘을 제시하였다[4, 5], 꼬리표개선 알고리즘은 후보노드집합에 후보노드를 추가하는 방법에 따라 다양한 알고리즘이 제시되어 왔다. Pape와 Pallottino는 후보노드를 두개의 대기행렬로 나누어 성능을 향상시키는 방법을 제시하였다[10, 11].

Cherkassky가 1996년에 실험한 결과에 따르면, 노드주위에 네개의 링크가 연결되는 단순 그래프에서는 꼬리표개선 알고리즘 계열이 꼬리표설정 알고리즘 계열보다 성능이 뛰어났으나 그래프의 연결상태가 무작위로 복잡해지거나 링크의 거리가 급격히 변하는 그래프에서는 꼬리표개선 알고리즘 계열이 연산시간을 측정할 수 없을 정도로 성능이 저하되는 결과를 도출하였다. 직접적인 원인은 탐색 완료된 집합에 속하는 노드가 후보노드에 재진입하는 회수가 많아지기 때문이다. 또한 재진입하는 후보노드가 정렬방법에 따라 결과적으로 누적거리가 작음에도 불구하고 정렬순서가 뒤에 존재함으로써 앞에서 선행 처리된 후보노드를 다시 후보노드집합에 재진입시키는 효과를 나타내기 때문이다. 본 논문에서 제시하는 다중 동적구간 대기행렬을 이용한 최단경로탐색 알고리즘은 재진입효과를 줄이기 위하여 후보노드집합의 특성을 이용한다. 후보노드집합에 속하는 노드들의 누적거리 값은 확률적으로 일정한 선형감소 형태를 가진다는 점이다. 이런 특성을 고려하여 후보노드집합을 여러 개의 대기행렬로 관리하고, 후보노드집합의 크기와 누적거리 분포에 따라 범위를 변경해 가는 방법을 사용한다.

본 논문의 구성은 다음과 같다. 제2장에서는 후보노드집합의 특성을 파악하여 후보노드집합이 일정한 선형감소 형태를 나타냄을 보이며, 제3장에서는 특성분석을 통한 결과를 토대로 다중 동적구간 대기행렬 알고리즘을 제시한다. 제4장에서는 비교실험을 통한 알고리즘의 성능을 분석하고 결론을 제시한다.

2.후보노드집합(Candidate Node Set : LIST)의 특성

꼬리표설정 알고리즘과 꼬리표개선 알고리즘을 구별하는 요소이면서 성능에 가장 영향을 많이 미치는 부분이 후보노드집합의 관리 방법이다. 기존에 제시된 알고리즘들은 후보노드집합의 특성을 분석하여 제시되지 않았고, 제시된 알고리즘들은 여러가지 임의의 실험조건에서 성능이 평가되었다. 이런 알고리즘 개선방법은 성능을 전적으로 실험결과에만 의존하여 알고리즘의 객관적인 예측을 하기 어렵다. 본 연구에서는 전체 그래프에 존재하는 링크들의 분포에 따라 후보노드집합의 특성을 구체적으로 분석하여 보다 효율적인 알고리즘을 제안하려 하며, 기존에 제시된 알고리즘의 일반적인 그래프 상황에서 성능예측을 도울 수 있도록 한다.

Cherkassky, Goldberg, Radzik은 최단거리탐색 알고리즘 비교를 위하여 링크의 임의 생성을 상수분포에서 하였고, 본 연구에서도 동일한 실험환경에서 수행하였다.[2] 그러므로 후보노드집합의 특성은 그래프의 링크가 상수분포일 경우를 가정하여 설명한다. 또한 모든 거리는 양수임을 가정한다.

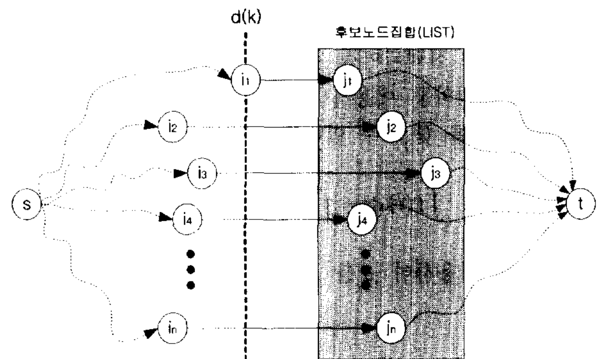
그래프에서 링크가 상수분포를 한다면 확률변수 a는 다음과 같다.

$$f(a) = \begin{cases} \dfrac{1}{A_h - A_l} & 0 \le A_l \le a \le A_h \\ 0 & \text{othrwise} \end{cases}$$

A_h : 상수분포의 최대값 매개변수

A_l : 상수분포의 최소값 매개변수

그래프에서 시점노드(s)에서 종점노드(t)까지 후보노드집합에서 최소거리를 가지는 노드를 후보노드로 결정하는 꼬리표설정 알고리즘방법에 의하여 연산을 진행시킨다. 꼬리표설정 알고리즘 방법으로 연산하는 이유는 노드가 재진입하는 효과를 분석에서 제외하기 위함이다. 재진입이 이루어지면 후보노드집합의 특성이 불특정하게 변하게 된다. 본 논문에서 제시하는 알고리즘을 수행하면 확률밀도함수의 0에 가까운 부분이 재진입 회수만큼 커지는 효과를 나타낸다. 본 논문에서 제시하는 알고리즘은 재진입이 발생하여 0에 가까운 부분의 확률이 높아져도 이 부분을 우선적으로 후보노드를 선택하게 됨으로 재진입의 회수를 줄이도록 하고 있다.



s : 시점노드

t : 종점노드

k : 가장 최근에 탐색 완료된 집합에 진입한 노드(i_1)

d(k) : 시점노드(s)에서 노드 k까지의 최단거리(실선)

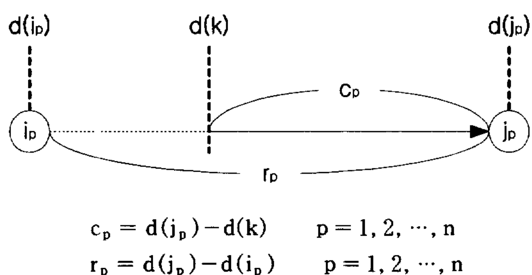
c_p : $d(j_p)-d(k)$ $p = 1, 2, \cdots, n$

(그림 1) 꼬리표설정 알고리즘에 의해 노드 k까지 연산된 그래프 상태

꼬리표설정 알고리즘으로 임의의 노드 k까지 연산진행이 완료되면, 시점노드에서 종점노드까지 최단거리로 진행하는 많은 후보 경로들이 존재하여 (그림 1)과 같은 그래프 상태가 된다. 이 그래프 상태에서 n개의 링크가 현재까지 계산된 거리 d(k)를 초과하는 상태에 있다고 하면, 그 링크의 분포 f(c)는 다음의 전개에 의하여 구할 수 있다.

이 상태에서 다음 연산진입 후보노드는 후보노드집합에 속하는 노드들 중d(k)에서 거리가 가장 가까운 노드가 된다.

(그림 2)에서 d(k)를 중심으로 c_p의 확률밀도함수를 구하



(그림 2) 노드 k까지 계산된 그래프에서 확률변수 c와 확률변수 r

기 위하여 다음과 같은 연산을 수행한다. 확률변수 r_p는 확률변수 a를 따르는 모집단에서 임의 추출하였으므로 확률변수 r_p는 확률변수 a와 동일한 확률분포를 따르게 된다. 거리 r이 결정되었다는 조건에서 거리 c의 조건부 확률밀도함수는

$$f(c \mid r) = \frac{1}{r} \quad 0 \le c \le r \qquad (1)$$

이다. 거리 r을 가지는 링크(i_p)가 d(k)이전에 거리 r보다 크지 않은 한도에서 일정하게 존재할 수 있기 때문이다. 즉, n번째 거리가 r이고, 상수분포를 확률변수 n-1개의 합으로 d(k)-r보다 크고 n개의 합으로 d(k)보다 작은 링크는 상수분포를 따른다.

한편, 확률변수 r의 확률밀도함수는 링크의 거리가 길수록 그 링크가 선택되어질 확률이 정비례하게 증가하게 된다. 확률의 정의를 이용하면 정비례 계수()를 찾으면, r의 확률밀도함수를 찾을 수 있다.

$$\int f(r)\,dr = \int_{A_l}^{A_h} \frac{1}{A_h - A_l}(\alpha)\,dr = 1 \qquad (2)$$

식 (2)를 이용하여 α를 구하면 식 (3)과 같은 f(r)이 계산되어 진다.

$$f(r) = \frac{2}{{A_h}^2 - {A_l}^2}\, r \quad A_l \le r \le A_h, \quad \alpha = \frac{2}{A_h + A_l} \qquad (3)$$

c와 r의 동시확률밀도함수는 식 (4)와 같다.

$$f(c, r) = f(c \mid r) f(r) = \frac{2}{{A_h}^2 - {A_l}^2} \quad A_l \le r \le A_h, 0 \le c \le r \qquad (4)$$

이를 각 두개의 영역으로 나누어 연산하여 c의 확률밀도함수를 식 (5)와 식 (6)을 통하여 구할 수 있다.[12]

i) $0 \le c \le A_l$

$$f(c) = \int f(c, r)\,dr = \int_{A_l}^{A_h} \frac{2}{{A_h}^2 - {A_l}^2}\,dr = \frac{2}{A_h + A_l} \qquad (5)$$

ii) $A_l \le c \le A_h$

$$f(c) = \int f(c, r)\,dr = \int_{c}^{A_h} \frac{2}{{A_h}^2 - {A_l}^2}\,dr$$

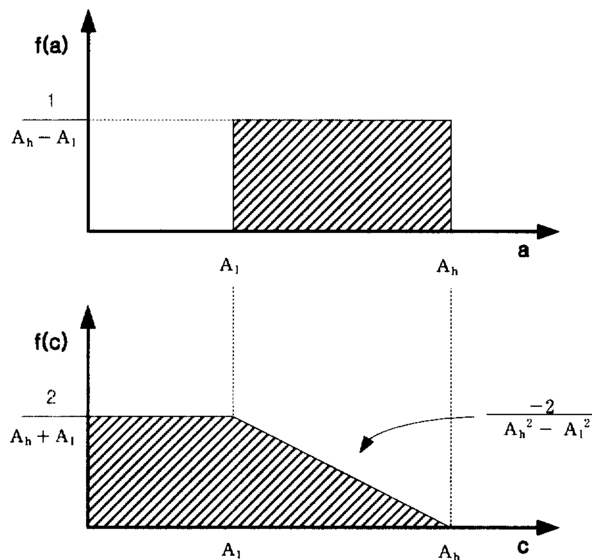
$$= \frac{-2}{{A_h}^2 - {A_l}^2}\, c + \frac{2A_h}{{A_h}^2 - {A_l}^2} \qquad (6)$$

결과적으로 (그림 3)에서 그래프는 링크가 가지는 상수분포는 연산과정 중의 후보노드집합에 속하는 누적거리 값들의 분포가 A_l에서부터 A_h까지 선형적으로 감소하는 확률분포함수를 따르게 됨을 알 수 있다.

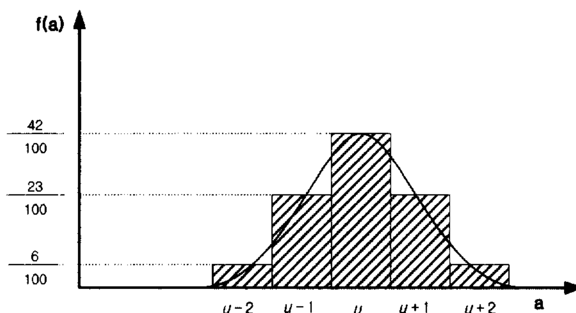
그래프에서 링크가 정규분포를 한다면 확률변수 a는 다음과 같다.

$$f(a) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(a-\mu)^2}{2\sigma^2}} \qquad -\infty \le a \le \infty$$

일반정규분포에서 확률변수 a는 음의 값을 가질 수 있다. 그러나, 링크값은 음의 값을 가질 수 없다고 초기에 정의하였으므로 실제로 링크가 정규분포를 따를 수는 없다. 또한 정규분포에서 정비례계수 α를 직접 계산으로 구할 수도 없다. 그럼에도 불구하고 일반적인 링크의 확률분포를 정규분포로 하려는 이유는 확률변수를 명확히 알 수 없는 경우에는 정규분포를 많이 가정하고, 다른 분포들은 경우의 수가 진행될수록 정규분포에 근사하게 되는 경우가 매우 많기 때문이다.



(그림 3) 확률변수 a와 확률변수 c의 확률밀도함수



(그림 4) 이산형으로 나타낸 확률변수 a의 확률밀도함수

링크가 정규분포에 가까운 근사정규분포를 따르는 경우

에 후보노드집합의 특성을 파악하기 위하여 표준정규분포에서 z값을 1단위의 이산형으로 나타낸 뒤 후보노드집합의 그래프를 구하도록 한다.

확률변수 a의 확률밀도함수를 이산형으로 나타내면 (그림 4)와 같이 나타난다.

거리 l이 결정되었다는 조건에서 거리 c의 이산형 조건부 확률밀도함수는

$$p(c \mid r=\mu-2)=\frac{1}{\mu-2} \quad c=1,2,3,\cdots,\mu-2$$

$$p(c \mid r=\mu-1)=\frac{1}{\mu-1} \quad c=1,2,3,\cdots,\mu-1$$

$$p(c \mid r=\mu)=\frac{1}{\mu} \quad c=1,2,3,\cdots,\mu$$

$$p(c \mid r=\mu+1)=\frac{1}{\mu+1} \quad c=1,2,3,\cdots,\mu+1$$

$$p(c \mid r=\mu+2)=\frac{1}{\mu+2} \quad c=1,2,3,\cdots,\mu+2$$

이다. 한편, 확률변수 r의 정비례계수를 구하면

$$\sum p(r)=\frac{\mu-2}{5\mu}\times\frac{6}{100}\times\alpha+\frac{\mu-1}{5\mu}\times\frac{23}{100}\times\alpha+\frac{\mu}{5\mu}\times\frac{42}{100}\times\alpha+\frac{\mu+1}{5\mu}\times\frac{23}{100}\times\alpha+\frac{\mu+2}{5\mu}\times\frac{6}{100}\times\alpha=1$$

$\alpha=5$가 되고, P(r)은

$$p(r=\mu-2)=\frac{6(\mu-2)}{100\mu},\quad p(r=\mu-1)=\frac{23(\mu-1)}{100\mu},$$

$$p(r=\mu)=\frac{42(\mu)}{100\mu},\quad p(r=\mu+1)=\frac{23(\mu+1)}{100\mu},$$

$$p(r=\mu+2)=\frac{6(\mu+2)}{100\mu}$$

$$P(c)=\sum_r p(c,r)=\sum_r p(c \mid r)p(r) \qquad (7)$$

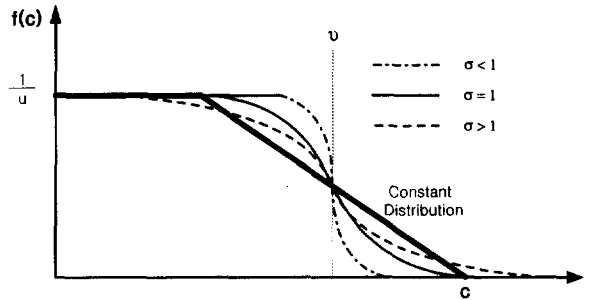
이 된다. 식 (7)에 의하여 p(c)를 구하면, c가 1부터 -2인 경우까지는 동일한 확률을 가지며, 그 이후의 값들 다음과 같이 감소하는 형태를 나타낸다.

$$p(c=1,2,3\cdots\mu-2)=\frac{6(\mu-2)}{100\mu}\times\frac{1}{\mu-2}+\frac{23(\mu-1)}{100\mu}\times\frac{1}{\mu-1}+\cdots+\frac{6(\mu+2)}{100\mu}\times\frac{1}{\mu+2}=\frac{1}{\mu},$$

$$p(c=\mu-1)=\frac{94}{100\mu},\quad p(c=\mu)=\frac{71}{100\mu},$$

$$p(c=\mu+1)=\frac{29}{100\mu},\quad p(c=\mu+2)=\frac{6}{100\mu}$$

이러한 결과는 정규분포에서 표준편차가 커지면 보다 넓은 영역에서 확률이 감소하기 시작하여 완만한 곡선을 이루게 된다. 한편, 상수분포와 비교하여 보았을 때, 평균값보다 작은 영역에서는 보다 큰 값을 나타내며, 그 이후 영역에서는 상대적으로 작은 값을 보여준다. 이러한 현상은 확률이 평균값에 가까운 정도에 따라서 폭이 더 늘어남을 볼 수 있다(그림 5).



(그림 5) 표준편차의 변화에 따른 c의 확률밀도함수의 변화

3. 다중 동적구간 대기행렬(Multiple Dynamic-Range Queue)

그래프에서 링크의 분포에 상관없이 후보노드집합의 확률분포는 평균보다 작은 전반부 부분에서 상수분포에 근사하며, 평균에서 최대값까지 단조감소해 가는 분포임을 알았다. 이러한 특성을 감안하여 꼬리표개선 알고리즘의 후보노드집합을 여러 개의 대기행렬로 나누어 관리하는 방법을 제안하고자 한다. 여러 개의 대기행렬로 나누어 관리하게 되면 후보노드집합에서 상대적으로 작은 거리 값을 가지고 있음에도 불구하고, 진입 시점이 늦어져 많은 노드들을 재진입시켜야 하는 문제점을 해소할 수 있다. 실제로 꼬리표개선 알고리즘 계열이 꼬리표설정 알고리즘 계열과 비교하여 성능이 많이 떨어지는 이유이기도 하다.

한편 그래프는 시점노드에서 시작하여 종점노드까지 탐색해 가면서 후보노드집합의 크기가 계속 변하게 된다. 일반적으로는 초기 그래프 연산에서는 급격히 증가하고, 후반부에서는 반대로 급격히 감소하는 특성을 나타낸다. 기존의 많은 알고리즘들도 후보노드집합 크기 변화에 대응하는 알고리즘은 극히 미비하며 DIKBD(Double Buckets)와 같은 일부 버킷계열 알고리즘에서 초반부와 후반부에 하위 버킷을 확인하지 않는 이점을 살리고 있을 뿐이다. 이러한 후보노드집합 크기의 변화에 대응하여 계산처리 속도를 향상시키기 위하여 대기행렬이 관리하는 구간의 범위를 후보노드집합의 크기 변화에 따라 구간을 변경하도록 하였다.

후보노드집합을 관리하는 기본적인 관리방법은 연산에 실질적으로 참여하는 후보노드의 수를 일정한 수로 유지하여 성능을 향상시키는 방법을 사용한다. 후보노드집합을 관리하는 기본 단위는 대기행렬이며, 그 대기행렬은 상위범위(Upper Range : UR)대기행렬, 하위범위(Lower Range : LR) 대기행렬, 별도의 대기행렬(Overflow Bag)로 이루어져 있다.[그림 6] 첫 번째 상위범위 대기행렬과 두 번째 상위범위 대기행렬은 실질적으로 존재하는 대기행렬이 아니며, 여러 개의 하위범위 대기행렬로 구성되어 있다는 상징적인 의미를 가진다. 후보노드집합에서 누적거리 값이 작은 노드

들이 첫 번째와 두 번째 상위범위 대기행렬에 속한다. 그 외의 모든 후보노드들은 별도의 대기행렬에 속하게 된다. 별도의 대기행렬은 상징적인 하나의 대기행렬을 의미하며, 여러 개의 상위범위 대기행렬로 구성되어 있다.

하위범위 대기행렬은 첫 번째와 두 번째 상위범위 대기행렬을 실질적으로 여러 개의 대기행렬로 나누어 사용하는 데 적용되어 진다. 하위범위 대기행렬을 사용하는 이유는 최단경로탐색 알고리즘 연산시에 후보노드 선택에 참여하는 후보노드의 수를 적정한 수준으로 유지하는 역할을 담당한다. 즉 첫 번째와 두 번째 상위범위 대기행렬에 속하는 후보노드의 수가 증가하면 그에 해당하는 하위범위 대기행렬의 수를 늘려 하나의 하위범위 대기행렬에는 일정한 수의 후보노드가 속할 수 있도록 조정하는 것을 의미한다.

별도의 대기행렬은 후보노드집합에 속하면서 누적거리 값이 매우 큰 노드는 최단경로탐색 연산에서 후보노드로 미리 선택되어 다른 노드들이 재진입하는 효과를 차단하기 위하여 사용되어 진다. 즉, 별도의 대기행렬을 두어 관리하는 이유는 링크의 길이가 상당히 큰 값이 존재하는 경우에 상위범위가 변경되는 시점에서 재진입 여부를 계속 판단해야 되기 때문이다. 만약 이 부분의 연산량이 많아지게 되면 연산성능이 급격히 나빠지게 된다. 또한 별도의 대기행렬을 여러 개의 상위범위 대기행렬로 나누어 관리하는 이유는 하나의 상위범위 대기행렬 연산이 완료되고 새로운 상위범위 연산을 위해 별도의 대기행렬에서 다음 상위범위에 해당하는 후보노드를 선택해 내야 한다. 이를 효과적으로 수행하기 위해서 사용되어 진다. 이는 별도의 대기행렬 범위에 속하는 노드의 수가 많아져 연산의 효율을 감소시키는 부분에 효과적으로 대응할 수 있다.

모든 t번째 상위범위 대기행렬(UR_t)의 길이는 초기 최단경로탐색 연산을 시작하는 시점에서 결정되어지며, 그래프 연산이 종료될 때까지 변하지 않고 일정한 길이로 유지된다. UR_t 길이를 결정하는 방법은 (그림 5)에서 평균 μ를 중심으로 결정한다. 후보노드집합은 확률밀도 함수를 0부터 μ까지 상수분포를 따르는 부분에 근사시킬 수 있기 때문이다.

한편 UR_t 가 결정된 상황에서 각 상위범위 대기행렬 구간마다 하위범위 대기행렬 범위를 결정해야 하는 문제가 발생한다. 그 하위범위 대기행렬의 개수에 따라 알고리즘의 성능은 영향을 많이 받는다. 다음 하위범위 대기행렬(n_{t+1})의 개수는 기본적으로 다음 상위범위 대기행렬에 후보노드의 수가 m_{t+1}개일 거라는 가정에 의하여 시작한다. 이 m_{t+1}개의 후보노드가 존재하는 상황에서 하위범위 대기행렬을 몇 개로 할 것인가를 결정해야 한다. 하위범위 대기행렬의 개수를 m_{t+1}보다 크게 하면 하위범위 대기행렬에는 후보노드가 존재하지 않는 빈 대기행렬이 많이 발생하여 효율이 떨어지고, m_{t+1}보다 매우 작게 하면 후보노드가 존재하지 않는 대기행렬은 존재하지 않게 되나 하위범위 대기행렬 내에서 후보노드 진입의 순서가 나빠져 다른 후보들이 재진입을 자주함으로 성능을 저해할 요소가 있다. m_{t+1}의 추정에 영향을 주는 요소는 현재까지 진행된 연산 결과에 따른 다음 상위범위 대기행렬에 포함된 후보노드의 수와 다음 상위범위 대기행렬을 진행하면서 새로 후보노드집합에 추가되는 후보노드 수의 곱하기로 추정할 수 있다. 연산 결과에 따른 다음 상위범위 대기행렬에 포함된 후보노드의 수는 연산이 진행되면서 알게 되는 값이고, 다음 상위범위 대기행렬을 진행하면서 새로 후보노드집합에 추가되는 노드의 수는 추정에 의존한다. 이 추정치(k)는 [전체링크의수 / 전체노드의수]를 사용한다. 본 실험에서는 다음 하위범위 대기행렬(n_{t+1})의 개수는 $m_{t+1}/(k*k)$의 성능이 가장 좋았다.

알고리즘의 수행절차는 다음과 같다.

[STEP 1] 첫 번째 LRMDRQ[0]에 시점노드를 추가한다. LRMDRQ의 수를 URMDRQ[0] = 1, URMDRQ[2] = k로 초기화 한다. 누적거리 = 0, LRMDRQDist, NextLRMDRQDist값을 초기화한다.

[STEP 2] 다음 URMDRQ의 추가노드 수를 0 으로 초기화 한다. 현재 URMDRQ에 대하여 대기행렬안에 후보노드가 존재하지 않는 경우까지 반복한다.

[STEP 2-1] 후보노드를 선택하고, 후보노드가 연산되었음을 알 수 있도록 visitid = 0으로 둔다.

[STEP 2-2] 후보노드 i에 연결된 모든 노드 j에 대하여 $d(j) > d(i) + c_{ij}$ 를 조사하여 만족시키면 [STEP 2-3]을 수행하고, 아니면 [STEP2-1]로 간다.

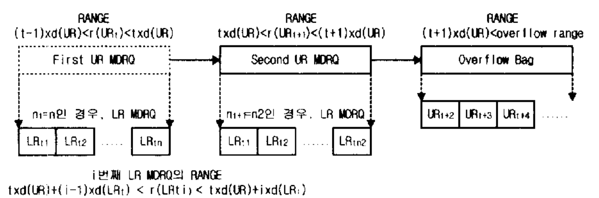
[STEP 2-3] j노드의 값을 $d(j) = d(i) + c_{ij}$ 로 새로 변경한다. j노드를 현재 누적거리 값의 차에 의하여 해당하는 URMDRQ를 찾는다. URMDRQ가 첫 번째이거나 두 번째이면서 이전순위 URMDRQ에 자신이 등록되어 있지 않으면, 그에 해당하는 LRMDRQ를 찾는다. 이에 해당하는 대기행렬에 새로운 후보노드를 추가하고, visitid는 해당 URMDRQ의 인덱스를 줌으로서 자신이 뒤에서 다시 후보노드가 되는 것을 방지한다. 두 번째 URMDRQ인 경우 다음 URMDRQ의 추가노드 수를 증가시킨다. URMDRQ가 세 번째 이상이고 이전순위 MDRQ에 자신이 등록되어 있지 않으면, 그에 해당하는 URMDRQ에 후보노드로 추가하고 visitid도 변경한다. [STEP2-1]로 간다.

[STEP 3] 누적거리 값과 URMDRQ를 증가시킨다.

[STEP 4] 모든 대기행렬에 후보노드가 존재하지 않으면 종료한다. 존재하면 계속진행한다.

[STEP 5] 다음 URMDRQ의 추가노드 수에 따라 그에 해당하는 LRMDRQ의 수를 결정한다.(nextURMDRQNum / (k*k)) LRMDRQDist, NextLRMDRQDist값을 연산한다.

[STEP 6] 별도의 대기행렬에서 처음 URMDRQ를 확장하여 LRMDRQ를 만들어 낸다.



[t번째 UR MDRQ가 변경되어진 시점에서 후보노드집합의 대기행렬 상태]

$n_{(t+1)}n2$: t+1번째 하위범위 대기행렬의 수

[t번째 상위범위 대기행렬의 변화과정과 t+1번 상위범위 대기행렬로 변해가는 후보노드집합의 대기행렬 변화과정]

UR : 상위범위 대기행렬

UR_t : t 번째 상위범위 대기행렬

d(UR) : 상위범위 대기행렬의 거리로 초기에 결정, 변경되지 않음. 즉, $d(UR_1) = d(UR_2) = \cdots = d(UR_t) = \cdots$

$r(UR_t)$: t 번째 상위범위 대기행렬의 범위를 나타냄

LR_{ti} : t 번째 상위범위 대기행렬에서 i번째 하위범위 대기행렬

$d(LR_t)$: t 번째 상위범위 대기행렬에서 하위범위 대기행렬의 거리로 t 번째 상위범위 대기행렬 안에서 변경되지 않음. 즉, $d(LR_{t1}) = d(LR_{t2}) = \cdots = d(LR_{ti}) = \cdots$

$d(LR_{ti})$: t 번째 상위범위 대기행렬에서 i번째 하위범위 대기행렬의 범위

n_t : t 번째 상위범위 대기행렬에 해당하는 하위범위 대기행렬의 수

(그림 6) 다중 동적구간 대기행렬(Multiple Dynamic-Range Queue)의 구조

다중 동적구간 대기행렬(Multiple Dynamic-Range Queue) 알고리즘은 (그림 7)과 같다.

```
typedef struct NODE_ {
   int      id ;         // Node ID
   double   x ;          // Node Position X
   double   y ;          // Node Position Y
   int      previd ;     // Previous ID for Optimal Path
   int      visitid ;    // Visited Tag for Optimal Path Computation
   long     nd ;         // Current Node Distance for Optimal Path
                            Computation
   int      an ;         // The number of Link From this Node To
                            Another Node
   ARC      arc[100] ;   // The pointer of Link From this Node To
                            Another Node
} NODE ;

BOOL QUEUENULL() ;          // Queue가 비었는지 확인하는 함수
LONG QUEUERemove() ;        // Queue의 처음노드를 꺼내는 함수
void QUEUEInsertTail(LONG i) ; // Queue의 꼬리에 노드를 추가하는 함수

algorithm Multiple Dynamic-Range Queue
{
   int k = 4 ;
   int LRMDRQNum[2] = { 1, k } ;
   int mu = 5000 ;   /* 평균 */
   int epack = 0 ;
   int opack = 1 ;

   int cIdx = nStart ; /* 시점노드 : nStart 종점노드 : nEnd */
   m_Node[cIdx].nd = 0 ;

/* 이미 연산이 완료된 노드를 처리하지 않기 위해 저장하는 UR
   MDRQ(t)값 */
m_Node[cIdx].visitid = 0 ;

MDRQPtr[epack][0]->QUEUEInsertTail(nStart) ;

long curdist = 0 ; /* 현재까지 진행된 누적거리 값 */

/* 첫 번째 UR MDRQ에 해당하는 LR MDRQ 범위 */
long LowerRangeMDRQDist = mu/ LRMDRQNum[epack] + 1 ;
/* 두 번째 UR MDRQ에 해당하는 LR MDRQ 범위 */
long NextLowerRangeMDRQDist = mu/ LRMDRQNum[opack] + 1 ;
int idx, div, sdiv, tag, nextURMDRQNum, t = 0, maxt = 0 ;

while (1) {
  nextURMDRQNum = 0 ; /* 다음 UR MDRQ추가되는
                          후보노드의 수 */
  for( i = 0 ; i< LRMDRQNum[epack] ; i++ ) {
    while ((cIdx = (int)MDRQPtr[epack][i]->QUEUERemove()) != -1 ) {
      m_Node[cIdx].visitid = 0 ;

      // Next Node Searching
      for (int k = 0 ; k<m_Node[cIdx].an ; k++) {
        idx = m_Node[cIdx].arc[k].td ;
        if (m_Node[cIdx].nd + m_Node[cIdx].arc[k].ad <
                               m_Node[idx].nd) {
          m_Node[idx].nd = m_Node[cIdx].nd +
                               m_Node[cIdx].arc[k].ad ;
          m_Node[idx].previd = cIdx ;
          div = (m_Node[idx].nd - curdist)/mu ;
          if( div < 1 ) { /* First UR MDRQ에 진입 */
            if( m_Node[idx].visitid > t ) {
              sdiv = ((m_Node[idx].nd - curdist)%mu)/
                       LowerRangeMDRQDist ;
              if( sdiv < i )
              sdiv = i ;
              MDRQPtr[epack][sdiv] ->
                              QUEUEInsertTail(idx) ;
              m_Node[idx].visitid = t ;
            }
          } else if ( div == 1 ) { /* Second UR MDRQ에
                                      진입 */
              if( m_Node[idx].visitid > t+1 ) {
                sdiv = ((m_Node[idx].nd - curdist)%mu)/
                        NextLowerRangeMDRQDist ;
              MDRQPtr[opack][sdiv] ->
                          QUEUEInsertTail(idx) ;
              m_Node[idx].visitid = t+1 ;
              nextURMDRQNum++ ;
            }
          } else {
            if( m_Node[idx].visitid > t+div ) { /* Overflow
                                      Bag에 진입 */
              OverflowBag[t+div]->
```

```
                    QUEUEInsertTail(idx) ;
                    m_Node[idx].visitid = t+div ;
                    if( maxt < t+div )
                       maxt = t+div ;
                 }
              }
           }
        } /* for loop */
     }/* while loop */
  }/* for loop */

curdist += mu ;
t++ ;

// Pack index change
if( epack == 0 ) { epack = 1 ;  opack = 0 ;
} else { epack = 0 ;  opack = 1 ;
}

// Queue가 비었는지 확인하는 절차
tag = 0 ;
for( i = 0 ;  i< LRMDRQNum[epack] ;  i++ ) {
   if( !MDRQPtr[epack][i]->QUEUENULL() ) {
       tag = 1 ;
       goto OUTPOSITION ;
   }
}
for( j = t ;  j<=maxt ;  j++ ) {
   if( !OverflowBag[j]->QUEUENULL() ) {
       tag = 1 ;
       goto OUTPOSITION ;
   }
}

OUTPOSITION :
   if( tag == 0 )
       break ;  /* Algorithm End */

   // 다음 UR MDRQ의 수를 예측, 다음 LR MDRQ개수를 결정
   LRMDRQNum[opack] = nextURMDRQNum/(k*k) ;
   if(LRMDRQNum[opack] < 1 )
       LRMDRQNum[opack] = 1 ;
   if(LRMDRQNum[opack] >= 1000)
       divi LRMDRQNum der[opack] = 999 ;
   LowerRangeMDRQDist = mu/ LRMDRQNum[epack] + 1 ;
   NextLowerRangeMDRQDist = mu/ LRMDRQNum[opack] + 1 ;
   // Overflow Bag의 처음 MDRQ를 LR MDRQ로 expansion함
   while ((cIdx = (int)OverflowBag[t+1]->QUEUERemove()) != -1) {
       if( m_Node[cIdx].visitid == t+1 ) {
             sdiv = (m_Node[cIdx].nd - curdist -
                    mu)/NextLowerRangeMDRQDist ;
             MDRQPtr[opack][sdiv]->QUEUEInsertTail(cIdx) ;
       }
    }
} /* Algorithm */
```

(그림 7) 다중 동적구간 대기행렬(Multiple Dynamic Range Queue) 알고리즘

4. 실험방법 및 실험결과

본 실험은 Pentium II 400Mhz CPU프로세서, 메모리 128M 바이트의 하드웨어와 Windows 2000의 소프트웨어 환경에서 실험하였다. 비교대상 알고리즘과 다중 동적구간 대기행렬 알고리즘은 C++언어로 코딩, Visual C++ 6.0으로 컴파일 하였으며, 컴파일러의 최적화 옵션은 사용하지 않았다.

기본적으로 비교대상 알고리즘은 꼬리표설정 알고리즘과 꼬리표개선 알고리즘의 대표적인 알고리즘과 실제적으로 성능이 우수하다고 평가되어 있는 알고리즘들을 선정하였다. 본 실험에서 비교대상 알고리즘으로 BFP, TWO-Q, Thresh, GOR, DIKH, DIKBD, HOT2를 사용하였다<표 1>.

<표 1> 실험에서 비교된 알고리즘의 요약설명

이 름	요 약 설 명
BFP	최상위 대기행렬값과 비교하는 Bellman-Ford-Moore 알고리즘 [4, 5]
TWO-Q	Pallottino의 Two Queue 알고리즘 [11]
Thresh	Glover의 Threshold 알고리즘 [12]
GOR	위상순서 탐색 알고리즘 [14]
DIKH	k-array Heap 알고리즘 [13]
DIKBD	Double Bucket 알고리즘 [2]
HOT2	Heap-On-Top에서 버킷의 깊이가 2인 알고리즘 [9]

실험을 위하여 그래프를 다양한 형태로 발생시켜 알고리즘을 비교하였다. 기본적으로 링크 거리값은 상수분포 [0, 10000]를 사용하였으며, 표준정규분포를 따르는 실험도 일부 병행하였다. 그래프는 기본적으로 정방형으로 노드를 배열한 후 이웃한 노드에 링크를 연결시키도록 하였다. 그래프는 노드를 증가시키면서 가장 먼 두개의 노드의 최단경로를 탐색하는 시간을 측정하였다. 처리시간이 1분을 초과하는 경우는 DNF로 처리하여, 시간을 측정하지 않았다. 그래프의 링크의 연결복잡도에 따라서 다음과 같은 그래프 환경에서 실험을 통해 알고리즘을 비교하기 위해 <표 2>와 같이 그래프의 성질을 변형해 가면서 실험을 반복하여 시간을 측정하였고, 결과의 객관성을 부여하기 위하여 각 그래프마다 생성용 임의변수를 변경해 가면서 10회씩 반복 실시하였다.

<표 2> 실험을 위해 사용된 그래프 발생방법에 대한 설명

이 름	요 약 설 명
Basic grid square	정방형 그래프 구조의 기본형태
RandomLong grid square	임의 링크연결, 링크의길이는 상대적으로 길
Random grid	링크연결을 임의로 발생시켜 연결함
Basic grid square double	Basic grid square에서 링크연결을 두배로 늘림
Random-Long grid double	링크연결을 임의로 발생시키고, 거리를 두배로 늘림
Stan-dev grid square	Basic grid square에서 링크의 분포를 정규분포로함
Random grid density	Random grid에서 링크의 수를 노드의 수를 제곱으로 늘림

실험의 결과는 <표 5>에서부터 <표 10>까지로 나타났다. 각각의 처리시간은 동일한 노드와 링크의 수를 가지고 링크의 길이와 링크의 연결을 변경해 가면서 10회 반복하여 평균을 나타낸 표이다. 동일환경에서 링크의길이와 연결이 임의로 변경한 상태에서 단위시간당 표준편차(μ/σ) 가 0.06이하로 나타나, 임의변수는 실험결과에 거의 영향을 주지 못한 것으로 나타났다.

<표 3>은 본 실험 구현방법의 타당성을 보여주기 위하여, Cherakassky가 실험한 기본 환경과 본 실험의 기본환경의 처리시간을 비교한 표이다. 실험에 사용된 그래프는 매우 유사하며, 하드웨어와 소프트웨어 환경은 다소 차이가 난다. 비교는 본 실험의 처리시간결과를 Cherkassky 실험결과 처리시간으로 나누어 상대적으로 비교하였다. 비교값들은 BFP와 DIKH를 제외하고 0.2에서 0.3사이에 대체적으로 존재하여 알고리즘 구현의 문제점은 없는 것으로 판단하였다. 단, BFP 결과를 보면 BFP 알고리즘 처리시간이 다른 알고리즘보다 상당히 우수하게 나타나는 데, 그 이유는 본 연구에서는 BFP를 구현할 때 Modified Label Correcting을 기본으로 구현하였지만, Cherkassky는 General Label Correcting방법을 사용한 것으로 추정되어 지기 때문에 결과값이 상당한 차이를 나타낸다. 그리고, DIKH가 DIKBD보다 대체적으로 우수한 형태를 보이는 데, 그 이유는 본 실험에서는 Heap의 속도를 가장 극대화 할 수 있는 Array구조로 구현하여 다른 알고리즘보다 비교값이 낮게 나온것으로 추정되어 진다[2].

〈표 3〉 본 실험[Basic Grid Square]과 Cherkassky[Simple Grid Square]의 처리시간비교

(Nodes/Links)	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD
본실험(1048576/4190208) 처리시간	1.36	1.37	1.41	2.48	1.92	1.62
Cherkassky실험(1048577/3145728) 처리시간	231.33	4.48	7.02	7.18	16.28	8.90
본실험처리시간 / Cherkassky처리시간(비율)	0.005	0.30	0.20	0.34	0.12	0.18

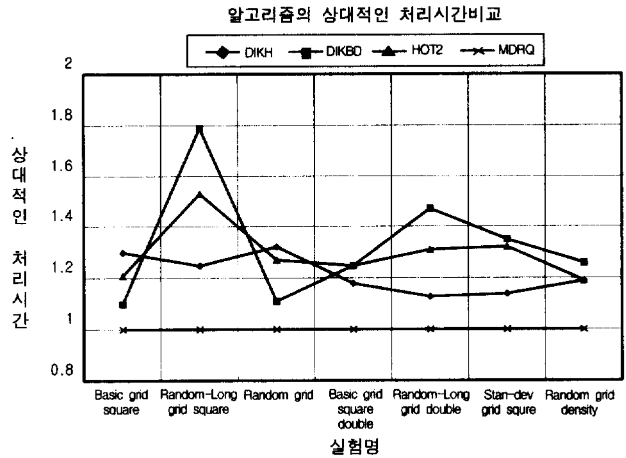
꼬리표개선 알고리즘 계열인 BFP, TWO-Q는 Basic grid square의 두 실험집합에서 성능이 우수함을 나타냈으나, 링크가 이웃하지 않은 노드에 연결되어 있는 경우에는 극단적으로 좋지않은 결과를 보여줬다. 이 결과는 기존의 연구에서도 이미 알려진 결과이기도 하다. 한편 Thresh와 GOR은 대체적으로 다른 비교 알고리즘보다 성능이 다소 떨어짐을 보여줬다.

DIKH, DIKBD, HOT2, MDRQ를 상세히 비교하기 위하여 실험마다 각 알고리즘의 처리시간을 MDRQ 처리시간으로 나누어 상대적인 처리시간을 비교한다. 각 실험에서 가장 노드가 많은 실험에 대해서만 비교하기로 한다. 그 비교 결과는 <표 4> (그림 8)로 나타난다.

〈표 4〉 노드가 가장 많은 실험별 알고리즘의 상대적인 처리시간비교

그래프 발생방법	DIKH	DIKBD	HOT2	MDRQ
Basic grid square	1.30	1.10	1.21	1.00
RandomLong grid square	1.25	1.79	1.53	1.00
Random grid	1.32	1.11	1.27	1.00
Basic grid square double	1.18	1.25	1.25	1.00
Random-Long grid double	1.13	1.47	1.31	1.00
Stan-dev grid square	1.14	1.35	1.32	1.00
Random grid density	1.19	1.26	1.19	1.00

본 연구에서 제시한 MDRQ 최단경로탐색방법은 실험결과와 같이 DIKH보다 20% 성능이 우수하게 나왔다. 꼬리표개선 알고리즘 계열의 BFP, TWO-Q가 취약성을 보여주는 Random-Long Grid Square에서도 성능이 저하되지 않았고, DIKH보다도 우수한 결과를 나타냈다. 또한 링크의 분포가 정규분포를 가져도 그 결과는 달라지지 않았다.



(그림 8) 노드가 가장 많은 실험별 알고리즘의 상대적인 처리시간비교

MDRQ 알고리즘의 복잡도는 다른 꼬리표개선 알고리즘 계열과 유사한 $O(m2^n)$으로 개선되지는 않았다. 그러나, 본 실험에서 여러 실험 환경에서 보편적으로 가장 우수한 성능을 보여주는 알고리즘임을 보였다. 또한 그래프의 특성과 후보노드집합의 특성을 연산이 진행되는 과정에서 계속 예

〈표 5〉 Basic Grid Square에서 알고리즘별 처리시간(10회 평균)

(단위 : sec)

Nodes/Links	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD	HOT2	MDRQ
4096/16128	0.00	0.00	0.01	0.01	0.00	0.01	0.01	0.00
16384/65024	0.02	0.01	0.02	0.03	0.04	0.04	0.04	0.02
65536/261120	0.07	0.07	0.07	0.16	0.10	0.13	0.12	0.08
262144/1046528	0.29	0.34	0.31	0.60	0.41	0.50	0.45	0.34
1048576/4190208	1.36	1.37	1.41	2.48	1.92	1.62	1.78	1.47

〈표 6〉 Random-Long Grid Square에서 알고리즘별 처리시간(10회 평균)

(단위 : sec)

Nodes/Links	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD	HOT2	MDRQ
4096/16128	0.03	0.03	0.02	0.03	0.01	0.01	0.01	0.00
16384/65024	0.58	0.63	0.24	0.47	0.04	0.05	0.05	0.03
65536/261120	5.73	4.64	1.84	3.77	0.21	0.33	0.28	0.16
262144/1046528	54.21	34.06	6.42	33.52	0.77	0.92	0.86	0.59
1048576/4190208	DNF	DNF	27.88	DNF	3.22	4.62	3.95	2.58

〈표 7〉 Random Grid에서 알고리즘별 처리시간(10회 평균)

(단위 : sec)

Nodes/Links	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD	HOT2	MDRQ
4096/16128	0.01	0.01	0.02	0.01	0.00	0.00	0.01	0.00
16384/65024	0.13	0.16	0.17	0.14	0.04	0.03	0.04	0.03
65536/261120	0.77	0.83	1.40	0.75	0.20	0.18	0.20	0.15
262144/1046528	3.62	3.56	4.95	3.05	0.73	0.67	0.72	0.54
1048576/4190208	17.63	14.61	14.89	14.23	3.12	2.62	2.99	2.36

<표 8> Basic Grid Square-d에서 알고리즘별 처리시간(10회 평균)

(단위 : sec)

Nodes/Links	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD	HOT2	MDRQ
4096/32256	0.00	0.00	0.00	0.01	0.00	0.01	0.01	0.01
16384/130048	0.03	0.02	0.02	0.04	0.02	0.04	0.04	0.02
65536/522240	0.08	0.09	0.09	0.21	0.12	0.15	0.15	0.12
262144/2093056	0.37	0.42	0.38	0.77	0.47	0.63	0.53	0.43
1048576/8380416	1.76	1.70	1.80	3.18	2.19	2.32	2.31	1.85

<표 9> Random-Long Grid Square-d에서 알고리즘별 처리시간 (10회 평균)

(단위 : sec)

Nodes/Links	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD	HOT2	MDRQ
4096/32256	0.07	0.07	0.04	0.06	0.01	0.01	0.01	0.01
16384/130048	1.05	1.13	0.50	0.75	0.06	0.07	0.07	0.06
65536/522240	9.36	8.51	2.79	5.32	0.36	0.43	0.41	0.26
262144/2093056	DNF	DNF	7.85	51.33	1.22	1.50	1.41	0.97
1048576/8380416	DNF	DNF	39.23	DNF	5.11	6.62	5.89	4.51

<표 10> Stan-dev Grid Square에서 알고리즘별 처리시간(10회 평균)

(단위 : sec)

Nodes/Links	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD	HOT2	MDRQ
4096/16128	0.00	0.00	0.00	0.01	0.00	0.01	0.01	0.01
16384/65024	0.03	0.02	0.02	0.02	0.04	0.04	0.04	0.02
65536/261120	0.12	0.08	0.08	0.11	0.13	0.16	0.15	0.10
262144/1046528	0.35	0.18	0.35	0.46	0.45	0.58	0.55	0.37
1048576/4190208	5.76	2.68	2.79	3.28	1.79	2.12	2.07	1.57

<표 11> Random grid density에서 알고리즘별 처리시간(10회 평균)

(단위 : sec)

Nodes/Links	BFP	TWO-Q	Thresh	GOR	DIKH	DIKBD	HOT2	MDRQ
256/65280	0.01	0.01	0.02	0.02	0.01	0.01	0.01	0.01
512/261632	0.06	0.03	0.04	0.04	0.03	0.04	0.04	0.03
1024/1047552	0.15	0.09	0.13	0.14	0.17	0.18	0.18	0.12
2048/4192256	0.71	0.23	0.45	0.56	0.57	0.65	0.64	0.20
4096/16773120	12.45	3.68	4.12	4.52	3.52	3.75	3.53	2.97

축하는 구조를 지니고 있으므로, 많은 그래프 형태에서도 상대적으로 안정적인 처리시간을 나타낸다. 본 논문에서 제시하는 알고리즘은 시간한계 특성은 향상이 없으나 일반적인 실험환경에서 우수한 결과를 보이는 알고리즘이다.

참 고 문 헌

[1] Narsingh Deo and Chi-yin Pang. "Shortest-Path Algorithms : Taxonomy and Annotation," Networks 14, pp.275-323, 1984.

[2] B. V. Cherkassky, A. V. Goldberg and T. Radzik, "Shortest paths algorithms : Theory and experimental evaluation," Mathematical Programming 73, pp.129-174, 1996.

[3] E. W. Dijkstra, "A note on two problems in connection with graphs," Numerical Mathematics 1, pp.269-271, 1959.

[4] R. Bellman, "On a routing problem," Quaterly Applied Mathematics 16, pp.87-90, 1958.

[5] L. R. Ford Jr., Network flow theory, Rand Co., pp.293, 1956.

[6] Floyd, R. W., "Algorithm 97 : Shortest path," Communications of ACM 5, pp.345, 1962.

[7] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," Journal of the ACM 34, pp.596-615, 1987.

[8] R. B. Dial, F. Glover, D. Karney and D. Klingman, "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees," Networks, 9, pp.215-248, 1979.

[9] Boris V. Cherkassky, Andrew V. Goldberg, Craig Silverstein, "Buckets, Heaps, Lists and Monotone Priority Queues," SIAM Journal on Computing 28, pp.1326-1346, 1999.

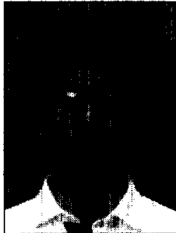
[10] Pape U., "Implementation and Efficiency of Moore Algorithms for the Shortest Root Problem," Mathematical Programming, 7, pp.212-222, 1974.

[11] Pallottino, S., "Shortest-Path Methods : Complexity, Interrelations and New Propositions," Networks, 14, pp.257-267, 1984.

[12] F. Glover, D. Klingman and N. Phillips, "A New Polynomial Bounded Shortest Path Algorithm," Operations Research Vol. 33, pp.65-73, 1985.

[13] Johnson D. B., "Efficient special-purpose priority queues," Proceedings of the 15th Annual Allerton Conference on Communication, Control and Computing, pp.1-7, 1977.

[14] R. B. Dial, "Algorithm 360 : Shortest path forest with topological ordering," Communications of the ACM 12, pp.632-633, 1969.



김 태 진

e-mail : w18375@korea.mot.com

1993년 고려대학교 산업공학과 졸업(공학사)

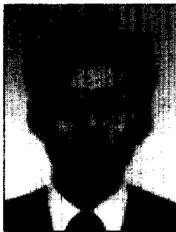
1995년 고려대학교 대학원 산업공학과 (공학석사)

1995년~2000년 ㈜팬택 중앙연구소 근무

1997년~현재 고려대학교 대학원 산업공학과 박사과정

2000년~현재 ㈜모토로라 코리아 테크놀러지 센터-CDMA 소프트웨어팀 선임연구원

관심분야 : ITS, NETWORK, LBS, CDMA, WAP, MPEG 등



한 민 홍

e-mail : mhhan@mail.korea.ac.kr

1964년 서울대학교 기계공학과 졸업(학사)

1973년 University of Minnesota 대학원 산업공학과(공학석사)

1981년~1982년 숭실대학교 산업공학과 조교수

1985년 Georgia Institute of Tech 대학원 산업공학과(공학박사)

1986년~1988년 Texas A&M University 산업공학과 조교수

1988년~1990년 포항공대 공과대학 산업공학과 부교수

1991년~현재 고려대학교 산업공학과 정교수

관심분야 : 지능형 자동차, ITS, 컴퓨터비전, 생산자동화