결함을 제한하는 자율안정 알고리즘을 이용한 중심과 반경을 찾는 시스템

김장환[1], 이충세[2]

A fault-containment self-stabilizing algorithm to find a center and radius

Jang-Hwan Kim[1], Chung-Sei Rhee[2]

요 약

Dijkstra가 분산시스템과 관련한 자율안정의 개념을 컴퓨터 과학에 처음 도입하였다. 그는 시스템이 초기 상태와 관계없이 유한한 수의 과정 안에 합리적인 상태에 도달한다는 것이 보증될 때 이러한 시스템을 "자율안정"이라고 정의하였다. 자율안정이 아닌 시스템은 불합리한 상태에 머무르게 된다. Dijkstra의 자율안정의 개념은, 처음에는 매우 협의의 단순한 응용이었는데, 분산시스템 상의 일시적인 결함 모델 하의 결함 허용으로의 정형적이며 일체화된 접근을 내포함을 증명하게 되었다. 본 논문에서는 자율 안정을 정의하고, 결함 억제와 관련해서 자율안정의 의미를 검사한다. 더욱이, 그래프의 중심과 중앙을 찾는 문제는 분산 환경의 통신 분야와 전송 분야에의 다양한 응용을 가진다. 본 논문은 그래프의 중심과 중앙을 찾기 위한 자율 안정 알고리즘을 보이고, 그 알고리즘의 정확성을 검증한다.

핵심어 : 자율안정, 결함 제한, 중심, 그래프

Abstract

Dijkstra introduced to computer science the notion of Self-Stabilization in the context of distributed system. He defined a system as Self-Stabilizing when "regardless of initial state, it is guaranteed to arrived at a legitimate state in a finite number of steps". A system which is not Self-Stabilizing may stay in an illegitimate forever. Dijkstra's notion of Self-Stabilization, which originally had a very narrow scope of application, is proving to encompass a formal and unified approach to fault tolerance under a model of transient failures for distributed system. In this paper, we define Self-Stabilization, examine its significance in the context of fault-containment. Furthermore, the problem of locating center and median of graphs has a variety of applications in the areas of transportation and communication in distributed system. This paper presents simple Self-Stabilizing algorithm for locating center and median of graphs and the correctness of the algorithm is proven.

Keywords : Self-Stabilization, fault-containment, center, graphs

접수일(2010년05월27일), 심사의뢰일(2010년05월28일), 심사완료일(1차:2010년06월15일, 2차:2010년06월24일)

게재일(2010년08월31일)

[1](교신저자)430-742 경기도 안양시 만안구 안양 8동 400-10 성결대학교 공과대학 교수
email: jhkim@sungkyul.ac.kr

[2]361-763 충북 청주시 흥덕구 성봉로 410 충북대학교 전자정보대학 컴퓨터공학부 교수
email: csrhee@cbu.ac.kr

1. 서론

최근 들어 분산 환경에 대한 중요성이 크게 인식되고 분산 환경 시스템의 연구가 급성장을 하고 있다. 전 세계가 인터넷이라는 큰 덩어리로 연결되어 수많은 정보를 공유하게 되었고, 클라우드 컴퓨팅 기술이 나날이 발전되어 가고 있다. 많은 시스템이 연결되어있는 분산 환경에서 구성원인 어느 시스템이 오류를 일으키게 될 때, 시스템이 외부의 어떠한 간섭 없이 자신과 이웃의 정보만을 이용하여 스스로 그 오류를 교정하여 안정된 상태로 복귀할 수 있다면 시스템을 운영 관리하는데 커다란 도움이 될 것이다. 시스템에서 오류가 발생했을 때, 외부의 간섭을 받지 않고 스스로 안정된 상태로 복귀하는 알고리즘은 Dijkstra에 의하여 처음으로 도입되었다.

Dijkstra는 시스템이 초기 상태와 관계없이 유한한 수의 과정 안에 합리적인 상태에 도달한다는 것이 보증될 때 이러한 시스템을 "자율안정"이라고 정의하였다. 자율안정이 아닌 시스템은 영원히 불합리한 상태에 머무르게 된다. Dijkstra가 자율안정이란 개념을 도입한 이후, Lamport가 Dijkstra의 연구를 결함허용 분야에서의 새로운 이정표라고 선언[1]함으로써 자율안정 알고리즘이 재조명하는 계기가 되었다. 그 후 활발한 연구가 진행되기 시작하였으며, 오늘날과 같이 분산 환경이 발달한 시대에 접어들면서 이 분야의 연구는 끊임없이 진행되고 있다.

분산 환경 시스템은 전역 메모리를 공유하지 않는 약-결합된 기계들의 집합으로 구성되어 있다. 네트워크를 연결하는 방법과 두 개의 기계들이 서로 통신을 하는 데 드는 시간에 따라서 각각의 기계는 전역상태를 부분적으로 파악할 수 있다. 대규모의 네트워크를 사용하는 분산 시스템에서 불특정한 시기에 여러 가지 형태의 오류가 발생할 수 있다. 이와 같은 오류는 기계를 손상시키거나 연결된 기계의 기능을 마비시킬 수도 있다. 오류가 발생했을 때, 오류에 대하여 대응방법이 없는 분산시스템 상의 알고리즘은 더 이상 정확한 기능을 수행할 수 없게 된다. 또한, 이러한 분산시스템에서 오류를 진단하고 교정하는 것은 매우 어렵다. 자율안정 알고리즘을 이용하면 오류가 발생 했을 때 시스템이 미리 지정된 방법에 의하여 오류를 교정하고, 스스로 합리적인 상태를 이룰 수가 있을 것이다.

이 논문에서는 분산 시스템에서 오류가 발생했을 때, 합리적인 상태로 스스로 안정화할 수 있는 알고리즘 (Self-Stabilizing algorithm) 을 분석하고 분산 환경의 통신 분야와 데이터 전송 분야에 매우 중요하게 사용되는 중심과 중앙을 찾는 문제에 자율 안정 알고리즘을 도입하여 분산 환경에서 트리의 중심과 중앙을 찾기 위한 Self-Stabilizing 알고리즘을 연구한다. 이 논문에서 Self-stabilizing 알고리즘을 "자율안정 알고리즘"이라 표기하기로 한다.

2. 자율안정의 정의

Dijkstra는 시스템이 외부의 간섭 없이 유한한 과정 안에 안정된 상태에 도달한다는 것이 보증

되고,, 안정된 상태에 도달한 후에는 계속하여 안정된 상태에 시스템이 머무를 때 이를 자율안정이라고 정의하였다[2].

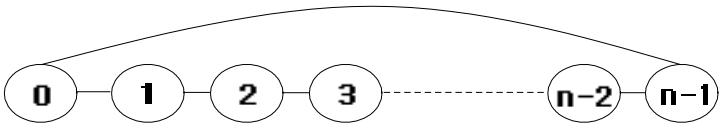
두 개 이상의 기계로 구성된 시스템에서, 각각의 기계를 프로세스라고 한다. 하나의 시스템은 두 가지 형태의 구성요소, 즉 프로세스와 이들 사이의 공유 메모리 또는 메시지 채널을 통한 연결로 구성된다. 시스템의 위상은 프로세스를 노드로 상호연결을 간선으로 표현하는 방향그래프로 나타낸다. 시스템을 구성하는 각각의 프로세스는 지역 상태를 갖는다. 시스템의 전역상태는 시스템을 구성하고 있는 각각의 프로세스의 지역 상태들의 합집합이다. 시스템의 행동은 상태들의 집합, 그들 간의 이행관계, 그리고 이행관계의 명확한 규칙들의 집합들로 구성된다. 전역상태를 만족하는 술어 P에 관한 시스템 S의 자율안정의 성질을 다음과 같이 정의한다[3].

만약 S가 다음 두 가지 성질을 만족한다면 S는 술어 P에 대한 자율안정이다.

- o 닫힘성 - P는 S의 실행에 관하여 닫혀있다. 즉, P가 S안에서 이루어졌다면 P는 오류가 될 수 없다.
- o 수렴성 - 임의의 전역상태에서 출발하여 유한한 수의 상태 변화를 하는 동안에 P를 만족하는 전역상태에 도달한다는 것이 보증된다.

P를 만족하는 상태를 합리적인 상태(legitimate state). P를 만족하지 않는 상태를 불합리한 상태(illegitimate state)라고 한다. 합리적인 상태와 불합리한 상태를 안전한 상태(safe state)와 불안전한 상태(unsafe state)라고 부르기도 한다.

2.1 Dijkstra의 모델



[그림 2-1] 링의 형태로 구성된 분산 시스템

[Fig. 2-1] Distributed system as a ring

1974년 Dijkstra는 분산 환경 시스템에서 자율안정이라는 개념[2]을 처음 도입하였다.

Dijkstra가 제시한 시스템의 모델은 링의 형태로 연결된 n 개의 프로세스로 구성되어있다. 각각의 프로세스는 k개의 상태를 갖고 있으며 자신과 자신의 이웃한 노드들에 의하여 읽혀지고 오직 자신만이 기록할 수 있다. Dijkstra는 특권(privilege)이라는 개념을 프로세스가 자신의 현재의 상태를 바꿀 수 있는 능력이라 정의하였다. 이 능력은 프로세스의 현재 상태와 연결된 이웃 프로세스의 상태에 근거하여 자신의 상태를 바꾸려고 할 때, 이 능력을 특권이라고 하며, 프로세스가 자신

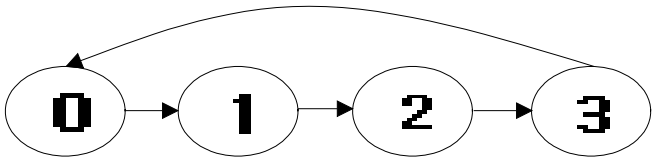
의 상태를 바꾸는 행위를 move라고 하며, 이 논문에서는 move를 "변화"이라고 표현한다. 더욱이 하나 이상의 프로세스가 특권을 가질 때 변화를 할 수 있는 프로세스를 선택하는 것은 시스템 전체를 총괄하는 어떤 중앙 디먼(central demon)에 의하여 결정된다. 합리적인 상태 하에서는 다음과 같은 성질들이 만족되어야 한다.

[성질 2.1]
[P1] 합리적인 상태에서는 특권을 갖는 프로세스가 정확이 하나만 존재하여야 한다.
[P2] 합리적인 상태로부터의 모든 변화는 시스템을 다시 합리적인 상태로 만든다.
[P3] 각각의 특권은 적어도 하나의 합리적인 상태에서 나타난다.
[P4] 두 개의 합리적인 상태가 존재할 때, 하나의 합리적인 상태를 다른 합리적인 상태로 바꾸는 변화의 열이 존재한다.

시스템이 초기의 상태와 무관하게, 그리고 매시간 다음의 변화를 위하여 선택되어지는 특권과 무관하게 유한한 횟수의 변화를 실행하는 동안 합리적인 상태에 도달하는 것이 보증될 때 그 시스템을 지율안정이라고 부른다. Dijkstra는 합리적인 상태를 하나의 프로세스가 특권을 가지는 상태로 표현하였다.

2.2 정형화된 네트워크와 비정형화 된 네트워크

자율안정에서의 문제점은 각각의 프로세스에서 실행되는 알고리즘이 정형화된 알고리즘인지 또는 비정형화 된 알고리즘인지에 대한 것이다. 네트워크에서 각각의 프로세스가 동일한 알고리즘을 사용하는 것은 바람직한 일이다.



[그림 2-2] 4개의 프로세스로 이루어진 링
[Fig. 2-2] Four processes ring

앞 절에서 Dijkstra의 모델은 적어도 하나 이상의 프로세스가 나머지 프로세스와는 다르게 작용한다고 가정하였다. 이와 같은 프로세스를 예외적인 프로세스라고 하며, 이 경우 알고리즘은 정형화 되어있지 않다. 분산 환경 시스템에서 자율안정 알고리즘이 정형화되는 것이 바람직하다[4].

알고리즘이 정형화되는 것이 바람직하지만, 대부분의 개발된 알고리즘은 적어도 하나의 예외적인 프로세스를 사용한다. 정형화된 알고리즘의 개발은 결코 쉬운 것이 아니지만, 다른 모델에서는

모든 프로세스가 동일한 알고리즘을 사용하는 정형화를 이루기도 한다.

2.3 Dijkstra 의 해

자율안정의 중요한 특성 중 하나는 각각의 프로세스들이 갖는 상태의 수이다. Dijkstra는 각각의 프로세스가 k개의 상태를 갖는 n개의 프로세스(0,1,…,n-1)들로 이루어진 방향 링 에 대한 세 가지 ($k \geq n$, k=4, k=3) 해를 제공하였다. Dijkstra의 모든 알고리즘은 다른 프로세스와 다르게 행동하는 예외적인 프로세스를 적어도 하나 이상 갖는다.

2.3.1 $k \geq n$ 인 경우의 알고리즘

Dijkstra의 천 번째 해는($k \geq n$) 시스템을 구성하는 프로세스의 수보다(n) 각각의 프로세스가 갖는 상태의 수 (k)가 크거나 같은 경우로 알고리즘[5]은 알고리즘2.1과 같다. 임의의 프로세스에 대하여, S는 자신의 상태, L은 왼쪽 이웃노드, R은 오른쪽 노드의 상태를 의미한다.

알고리즘2.1 $k \geq n$ 인 경우의 Dijkstra의 알고리즘

```
{예외적인 프로세스}
if L=S then S:=(S+1) mod K;
{다른 프로세스}
if L ≠ S then S:=L
```

알고리즘 2.1의 알고리즘은 단순하다. 그러나 이 알고리즘은 링의 크기에 종속되는 상태의 수를 필요로 한다.

2.3.2 k=3 인 경우의 해

Dijkstra의 알고리즘 중에 시스템을 구성하는 각각의 프로세스가 갖는 상태의 수가 3인 경우의 알고리즘[5]은 알고리즘 2.2와 같다. 각각의 기계들의 상태는 0, 1, 2만을 갖는다.

알고리즘2.2 k=3인 경우의 Dijkstra의 알고리즘

```
{최하위 프로세스(0)에 대하여}
if (S+1) mod 3= R then S:=(S-1) mod 3
{최상위 프로세스(n-1)에 대하여}
if L=R and(L+1) mod 3 then S:=(L+1) mod 3
{기타의 프로세스에 대하여}
if (S+1) mod 3 = L then S:=L
if (S+1) mod 3 = R then S:=R
```

[표 2-1] k=3인 경우의 Dijkstra의 알고리즘의 분석
[Table 2-1] Analysis of Dijkstra algorithm for k=3

프로세스0의 상태	프로세스1의 상태	프로세스2의 상태	프로세스3의 상태	특권을 갖는 프로세스	변화를 하는 프로세스
0	1	0	2	0,2,3	0
2	1	0	2	1,2	1
2	2	0	2	1	1
2	0	0	2	0	0
1	0	0	2	1	1
1	1	0	2	2	2
1	1	1	2	2	2
1	1	2	2	1	1
1	2	2	2	0	0
0	2	2	2	1	1
0	0	2	2	2	2
0	0	0	2	3	3
0	0	0	1	2	2

위의 표2.1은 Dijkstra의 3가지 상태를 갖는 프로세스에 대한 간단한 예를 보인 것이다. 이 시스템은 네 개의 프로세스(0, 1, 2, 3)로 구성된 링이라고 가정하자. 프로세스 0은 최하위 프로세스이고 프로세스 3은 최상위 프로세스이다. 표2.1의 마지막 열은 특권을 갖는 프로세스들 중에 변화를 하는 프로세스이다. 특권을 가진 프로세스의 수는 시스템에서 오직 하나만 존재 할 때 까지 계속 감소한다. 위의 표에서 최종적으로 특권을 갖는 프로세스가 시스템 안의 기계들을 0, 1, 2, 3, 2, 1, 0, …의 순서로 순회함을 알 수 있다. 앞에서 제시한 속성 P1, P2, P3, P4가 모두 만족되므로 시스템은 안정된다

3. 결함을 제한하는 자율안정 알고리즘

3.1 자율안정 알고리즘의 제한성

앞 절에서 자율안정에 대한 의미를 알아보고 Dijkstra의 모델에 대한 세 가지 해에 대하여 살펴보았다. 시스템이 스스로 합리적인 상태로 수렴한다는 특성은 분산시스템에서 요구되는 중요한 성질임에도 불구하고, 자율안정 알고리즘은 두 가지 문제로 인하여 현재의 분산 시스템에 적용하는 것은 매우 난해한 문제로 남아있다.

첫 번째 문제는 시스템이 임의의 초기상태로부터 합리적이 상태로 복구되는 동안에 시스템의 행동에 대한 어떠한 형태의 보증이 없다는 것이다. 자율안정 알고리즘은 단지 시스템이 어떤 합리적인 상태로 수렴하는 것만을 보증한다. 그러나, 시스템은 바람직한 상태로 수렴하는 동안에 임의로 행동한다. 이러한 사실은 시스템이 합리적인 상태로 수렴하는 동안에 비정상적으로 행동할 수

있음을 의미하며, 때로는 정상적인 시스템이 이에 연결도니 다른 시스템의 영향으로 비 정상정적인 행동을 할 수도 있다는 것을 의미한다. 이러한 사실은 시스템의 안정성 또는 신뢰성을 요구하는 많은 응용프로그램에서 허용할 수 없다.

비록 시스템의 하나의 구성요소에서 발생한 오류를 복구한다 하더라도 때로는 많은 양의 시간을 요구할 수도 있고, 때로는 시스템 전체를 오염시킬 수도 있다. 이와 같은 경우에 있어서 오류를 복구하는 시간은 종종 많은 구성요소의 오류를 복구하는 시간보다 클 수도 있다. 이러한 단점으로 인하여 실질적인 응용에 제약을 받고 있지만 시스템이 스스로 안정을 이룰 수 있다는 특성 때문에 자율안정 알고리즘에 대한 많은 연구가 수행되고 있다.

3.2 결함을 제한하는 자율안정 알고리즘

앞에서 제시한 자율안정 알고리즘의 문제점을 해결하기 위한 방안으로서 자율안정 뿐만 아니라 결함을 제한하는 특성까지도 갖춘 알고리즘이 필요하게 되었다. 결함제한(fault-containment)라는 개념은 적은 요소에서 발생하는 일시적인 오류로부터 시스템을 빠르게 복구하고, 오류의 영향을 효율적으로 제한하는 시스템의 능력을 나타낸다[7]. 결함제한은 시스템에서 발생한 일시적인 오류로 기인한 상태로부터 합리적인 상태로 수렴하는 동안 다음 두 가지 특성을 만족시켜야 한다.

- ▶ 시스템의 모든 구성요소에 의하여 변화되는 지역적인 상태의 총수는 작아야 한다.
- ▶ 결함이 있는 구성요소의 주위에 이는 구성 요소들이 부분집합들만이 자신의 지역적인 상태를 바꿀 수 있다.

바꾸어 표현하면 일시적인 오류에 의한 상태로부터 빠르게 수렴할 뿐만 아니라 오류의 영향이 시스템 전체에 미치지 않고 결함 있는 영역의 주위에 있는 구성요소들에만 제한된다. 자율안정과 결함제한의 정의를 만족시키는 알고리즘을 "결함을 제한하는 자율안정 알고리즘"이라 정의한다.

3.3 결함제한의 문제점

단일 구성요소에서 오류가 발생했다하더라도 결함제한과 자율안정을 결합하는 것은 매우 어렵다. 이와 같은 경우를 보여주는 예를 하나 살펴보자.

n개의 프로세스를 갖는 네트워크의 생성트리를 만드는 자율안정 알고리즘을 고려해보자. 각각의 프로세스들은 지역변수의 집합을 갖고 있다. 프로세스 i의 지역변수들은 i와 i에 이웃한 노드에 의해 읽혀질 수 있고, 오직 i에 의하여 기록될 수 있다고 가정한다. 그러므로 프로세스는 오직 자신과 이웃의 지역변수를 읽고, 자신의 지역변수에만 기록함으로서 네트워크상의 이웃과 통신을 할

수 있다.

4. 중심과 중앙을 찾기 위한 자율안정 알고리즘

그래프가 주어졌을 때, 그래프의 중심과 중앙을 찾는 문제는 분산 환경에서의 통신 분야와 데이터 전송 분야에서 많이 사용된다. 중심과 중앙을 찾는 문제에 자율안정 알고리즘을 적용할 때, 이러한 알고리즘은 일시적인 오류를 허용할 수 있을 뿐만이 아니라 동적으로 트리의 위상을 변화시킬 수도 있다.

$G=(V,E)$를 정점들의 집합 V와 간선들의 집합 E를 갖는 단순한 그래프라고 하자. 정점 $i \in V$의 이심률은 V안의 임의의 정점과 j에 이르는 거리 중 가장 큰 값을 의미한다. 이심률이 가장 작은 정점을 G의 중심이라고 부르다. 정점 $i \in V$의 값은 V의 모든 정점과 j에 이르는 거리의 합이다. 최소의 값을 갖는 정점을 G의 중앙이라고 한다.

분산 환경에서 공통으로 사용하는 자원을 중심이나 중앙에 위치시키는 것은 다른 곳에 위치시키는 것보다 자원을 공유하는데 드는 경비를 최소화 할 수 있기 때문에, 그래프의 중심과 중앙에 위치한 정점은 응용프로그램에서 중요한 의미를 갖는다. 그래프의 중심과 중앙의 위치에 관한 알고리즘은 통신 등에 많이 응용된다.

4.1 중심과 중앙을 찾는 알고리즘

알고리즘을 전개하기 위하여 다음과 같이 용어를 정의한다.

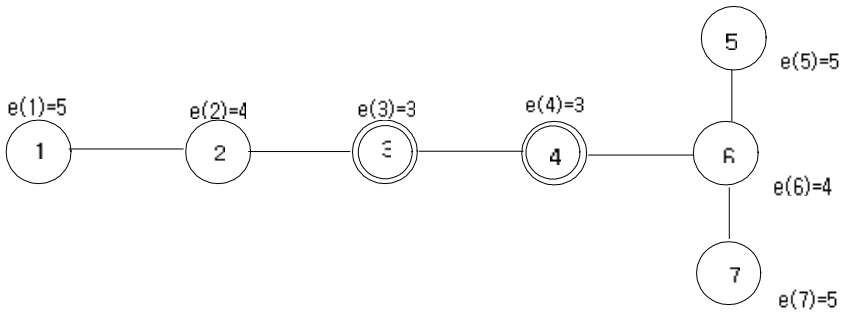
$d(i,j)$: 그래프 G에서 노드 i에서 노드 j에 이르는 최단거리.

$e(i) = \max\{d(i,j) \mid j \in V\}$: 노드 i의 이심률.

중심(G) = $\{ i \in V \mid e(i) \le e(j), \text{ for all } j \in V\}$:
그래프 G의 중심을 의미하며 최소의 이심률을 갖는 정점

$w(i)= \sum_{j \in V} d(i,j)$: 정점 i의 값

중앙(G) = $\{i \in V \mid w(i) \le w(j), \text{ for all } j \in V\}$:
그래프 G의 중앙을 의미하며 최소의 w(i)를 갖는 정점

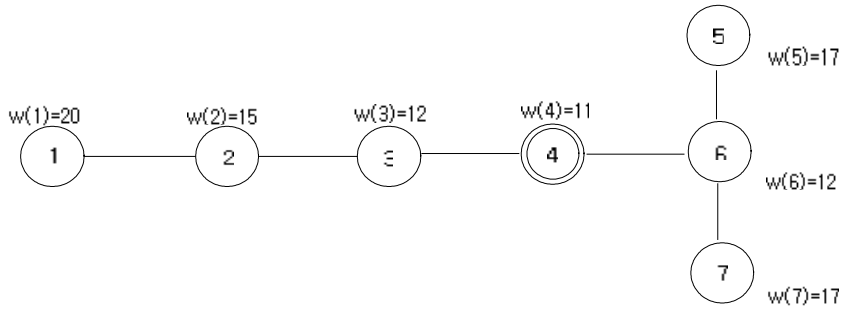


[그림 4-1] 트리에서 정점의 이심률과 중심

[Fig. 4-1] Eccentricity and center of vertices in tree

그림 4.1에서는 각각의 노드에서의 이심률을 나타내었다. 이 그래프에서 중심은 3번과 4번 두 개의 노드가 된다. 그림 4.2에서는 각각의 노드의 값을 나타낸다. 이 경우의 값이 ;가장 작은 노드 4가 중앙이 된다.

앞으로 정점들의 집합 V와 간선들의 집합 E를 갖는 트리 T=(V,E)로 문제를 제한한다. V={1,2,3, …,n} 이라 가정하자. 트리에서 중심과 중앙은 각각에 대하여 하나씩 존재하거나 또는 인접한 두 개의 중심이나 중앙이 존재한다는 것은 잘 알려진 사실이다.



[그림 4-2] 트리에서 정점의 값과 중앙

[Fig. 4-2] Vertex weight and center in tree

4.1.1 계산 모델

트리의 중심과 중앙에 위치에 관한 자율안정 알고리즘을 도입하기 전에 기초적인 계산 모델을 살펴보자. T에 있는 각각의 정점을 프로세스라고 가정하자. 각각의 프로세스는 이웃한 프로세스에 의하여 읽을 수 있고, 오직 자신에 의하여 기록할 수 있는 단일의 지역 변수를 유지한다.

프로세스 ;의 프로그램은 다음과 같이 표현한다.

G[1] → M[1] 또는 G[2] → M[2] 단,

▷ 각각의 조건 G[j](1 ≤ j ≤ 2)는 프로세스 $_i$의 지역변수와 그와 이웃한 프로세스의 지역변수에 관한 논리 함수이다.
▷ 각각의 변화 M[j](1 ≤ j ≤ 2)는 프로세스 $_i$의 지역변수를 수정하는 행동이다.
▷ 위의 문장은 모든 조건 G[j]가 거짓일 때까지 M[j]를 반복하여 실행하는 것을 의미한다.

각각의 변화 M[j](1 ≤ j ≤ 2)는 독자적인 행동을 한다고 가정한다. 각각의 프로세스에 속한 조건이 동시에 참이 될 수도 있다. 따라서 활동 가능한 조건을 가는 프로세스 중에서 임의로 하나를 선택하는 중앙 스케쥴러가 존재한다고 가정하자. 따라서 중앙 스케쥴러가 선택한 프로세스가 필요한 작업을 실행하는 동안 다른 프로세스는 대기 중이며, 이러한 작업이 끝났을 때 모든 프로세스는 다시 조건을 조사한다. 제안된 알고리즘은 자율안정이므로 변수의 초기화가 불필요하다. 더욱이, 기본을 이루는 네트워크는 위상 연결되어 있고, 사이클을 이루지 않는다는 조건하에서 변화가 허용된다. 일시적인 오류로 인하여, 프로세스는 이웃한 식별자에 대한 부정확한 정보를 일시적으로 갖는다. 그러나 오류가 일시적이므로 프로세스는 최종적으로 이웃한 노드를 정확하게 인식할 것이고, 시스템은 연속적으로 바람직한 상태로 수렴할 것이다.

4.1.2 알고리즘

알고리즘의 설명을 위하여 두 개의 함수를 도입한다. 각각의 정점 $_i$에 대한 함수로서, 정점들의 집합을 자연수의 집합으로 사상하는 h(i)와 정점들의 집합을 양의 정수로 사상하는 s(i)를 다음과 같이 정의한다.

h(i) : V → N
s(i) : V → N-{0}

이 때, 정점 $_i$∈V에 대한 h(i)는 가장 작거나 또는 두 번째로 작은 이심률을 갖는 두 정점을 루트로 갖는 트리에서 $_i$를 루트로 갖는 부분트리의 높이라 정의하고, s(i)는 가장 작거나 또는 두 번째로 작은 값을 갖는 두 정점을 루트로 갖는 트리에서 $_i$를 루트로 갖는 부분트리의 크기(정점의 수)라 정의한다. 이러한 정의에 부가하여 다음과 같은 개념을 사용한다.

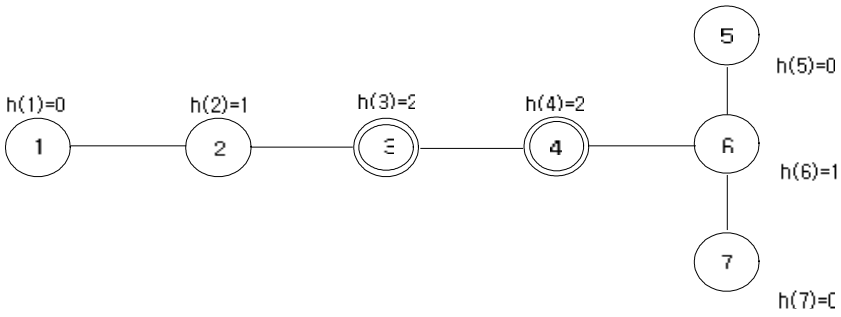
N(i)={j | (i ,j) ∈E} : 정점 $_i$와 이웃한 노드들의 집합
N_h(i)={h(j) | (i ,j) ∈E} : 정점 $_i$와 이웃한 h(i)의 집합
N_h^-= N_h(i) - {max(N_h(i))} : N_h(i)에서 최대값을 제외한 집합
N_s(i)={s(j) | (i ,j) ∈E} : 정점 $_i$와 이웃한 s(i)의 집합
N_s^-= N_s(i) - {max(N_s(i))}: N_s(i)에서 최대값을 제외한 집합.

정점 $_i$에 대한 h(i)를 높이조건이라고 정의한다.

$$h(i) = \begin{cases} 0 & \text{단노드일 경우} \\ 1 + \max(N_h(i)) & \text{단노드가 아닐 경우} \end{cases}$$

단, 여기서 max(N_h(i)) 는 N_h(i)의 최대값을 의미한다.

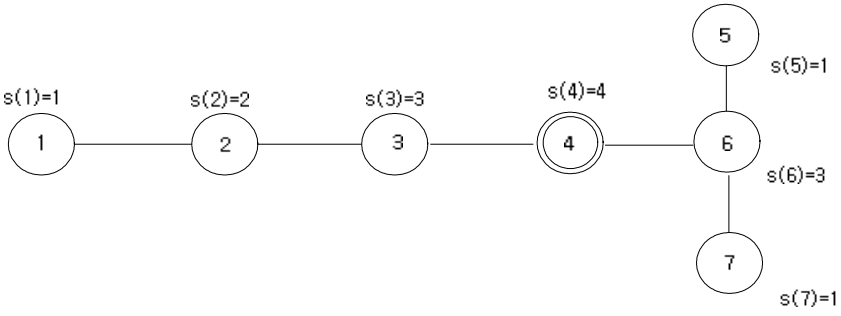
그림 4.1의 이심률에서 보듯이 가장 작은 이심률을 갖는 노드는 3과 4이므로, h(i)는 노드 3 또는 노드 4를 루트로 갖는 트리구조에서 구할 수 있다.



[그림 4-3] 트리에서 높이 조건을 만족하는 정점의 h(i)

[Fig 4-3] Vertex h(i) satisfying tree's height

그림4.3과 같이 h(i)를 구할 수 있는데, 각각을 살펴보면, 노드 3이 루트일 때, 하위노드는 1과 2이므로 노드 3의 높이는 2이다. 노드 3을 루트로 갖는 트리에서 노드 2는 부모 노드가 3이고 하위노드가 1이므로 노드 2를 루트로 갖는 부분 트리는 높이가 1이다. 따라서 h(2)=1이다. 또한 노드 4를 루트로 갖는 트리에서는 하위노드가 5,6,7이므로 h(4)=2이고, 노드 6의 하위노드는 5,7 이므로 h(6)=1이다. 또한 단노드 1, 5, 7 은 높이가 0이 된다. 여기서 알 수 있듯이 트리 T에 있는 모든 정점의 h(i)는 높이조건을 만족한다. 따라서, 가장 큰 h(i)를 갖는 정점이 트리 T의 중심이다. 또한 중심의 h(i)는 이웃 노드의 h(i)보다 크거나 같다는 사실을 알 수 있다. 그림 4.3에서 보듯이 최대의 h(i)를 갖는 노드는 3과 4, 즉 트리의 중심이 두 개다.



[그림 4-4] 트리에서 크기조건을 만족하는 정점의 s(i)

[Fig. 4-4] Vertex s(i) satisfying tree's weight

다음과 같은 정점 i의 s(i)을 크기 조건이라고 정의한다.

$$s(i) = \begin{cases} 1 & \text{단노드일 경우} \\ 1 + \sum (N_s(i)) & \text{단노드가아닐경우} \end{cases}$$

단, $\sum(N_s(i))$은 $N_s(i)$의 모든 원소의 합이다.

T에서 모든 정점의 s(i)가 크기조건을 만족한다면 가장 큰 s(i)를 갖는 노드가 트리T의 중앙이다. 그림4.4은 트리의 모든 정점의 s(i)가 크기 조건을 만족하는 것을 볼 수 있다. 최대의 s(i)를 갖는 정점 4가 트리에서 유일한 중앙임을 알 수 있다.

중심을 찾는 알고리즘과 중앙을 찾는 알고리즘은 알고리즘 4.1과 같다. 중심을 찾는 알고리즘의 두 문장은 정점 i에 의하여 높이조건이 만족됨을 알 수 있다. 따라서 중심을 찾는 알고리즘의 조건부가 거짓이면 T 안의 모든 정점의 h(i)는 높이조건을 만족한다. 이와 마찬가지로 중앙을 찾는 알고리즘의 조건부가 거짓이면 T 안의 모든 정점의 s(i)는 크기조건을 만족한다.

알고리즘 4.1 중심과 중앙을 찾는 알고리즘

```
{정점 i에 대하여 중심을 찾는 알고리즘}
[(i는 단 노드) ∧ (h(i) ≠0)
                    → h(i) := 0
or (i는 단 노드가 아님) ∧ (h(i) ≠ 1+max
(N_h^-(i))  → h(i) := 1+max(N_h^-(i))]

{정점 i에 대하여 중앙을 찾는 알고리즘}
[ (i는 단 노드) ∧ (s(i) ≠1)
                    → s(i) := 1
or (i는 단 노드가 아님) ∧ ( s(i) ≠
1+ ∑(N_s^-(i)) → s(i) := 1+ ∑(N_s^-(i))]
```

4.2 알고리즘의 정확성

알고리즘 4.1을 증명하기 위해, 순서 함수의 개념을 도입하자.

함수 f : V → N 은 만약 모든 정점 $i \in$V에 대하여 i의 이웃 노드 j가 적어도 하나 이상 존재하여 f(i) ≤ f(j)를 만족하면 함수 f는 순서함수이다. 즉, f(i)를 정점 i에서의 함수 값이라고 할 때, 다음의 결과는 쉽게 보일 수 있다.

[소정리 4.1] 만약 T안의 모든 정점들의 h(i)가 높이조건을 만족한다면, 함수 h는 순서함수이다.

[소정리 4.2] 만약 T안의 모든 정점들의 s(i)가 크기조건을 만족한다면, 함수 s는 순서함수이다.

증명 : T 안의 모든 정점이 높이조건을 만족하므로, 모든 $i \in T$에 대하여 i가 단노드이면 h(i)=0 이며 단노드가 아니면 h(i) := 1+max($N_h^-(i)$)이다. 따라서, 단노드가 아닌 $i \in T$에 대하여 $N_h^-(i)$에는 $h(k) \leq h(i)$를 만족하는 $k \in T$ 가 존재한다.

또한 k가 단노드가 아니라면 $h(t) \leq h(k)$인 t가 $N_h^-(k)$에 존재한다. 귀납법에 의하여 t가 단노드인 경우에도 $h(t) \leq h(k)$가 만족된다. 이를 바꾸어 표현하면, 모든 $i \in T$에 대하여 어떤 j가 존재하여 $h(i) \leq h(j)$를 만족한다. 따라서 함수 h는 순서함수이다.

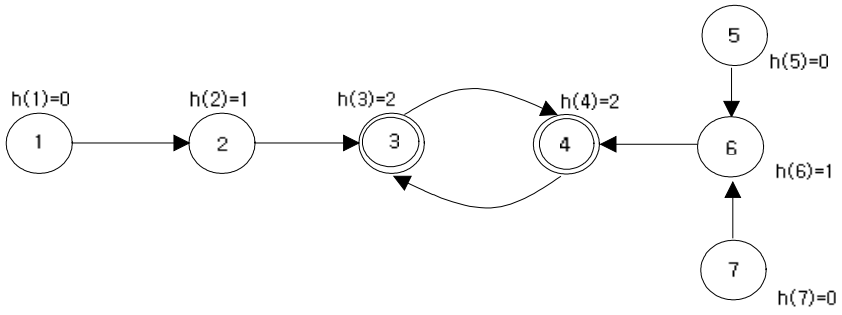
또한, T안의 모든 정점이 크기조건 s(i)를 만족하므로 h(i)와 마찬가지로 정의에 의하여 단노드가 아닌 $i \in T$에 대하여 $N_s^-(i)$에는 $s(k) \leq s(i)$를 만족하는 $k \in T$가 존재한다. 즉, 모든 $i \in T$에 대하여 어떤 j가 존재하여 $s(i) \leq s(j)$를 만족한다. 따라서 함수 s는 순서함수이다.

예를 들어 그림 4.3에서 보여준 트리에서 모든 정점의 h(i)는 높이조건을 만족한다. 각각의 정점에서 자신의 h(i)보다 크거나 같은 h(i)를 갖는 인접한 노드가 적어도 하나 이상 존재한다는 사실은 쉽게 알 수 있다. 그림 4.4에서도 비슷하게 트리 안의 모드 정점들의 s(i)가 높이조건을 만족함을 알 수 있고, 각각의 정점에서 자신의 s(i)보다 크거나 같은 s(i)를 갖는 인접한 노드가 적어도 하나 이상 존재한다는 사실은 쉽게 알 수 있다.

순서함수 f: $V \rightarrow N$ 가 주어졌을 때, 방향그래프(digraph) G(f)를 G(f) = (N(f), A(f)) 이라 정의하자. 단, 노드들의 집합을 N(f)=V라고 할 때, 전이집합 A(f)는 다음과 같이 정의하자

A(f) = { (i, j) | $j \in N(i)$ 이고 $f(j)=\max\{N_f(i)\}$}

따라서, 각각의 노드 $i \in N(f)$로부터 가장 큰 f(j)를 갖는 인접한 노드로 가는 간선이 존재한다. 만약 가장 큰 f(j)를 갖는 인접 노드들이 여러 개가 있다면 가장 큰 f(j)를 갖고 레벨의 값이 가장 큰 인접 노드를 선택함으로서 하나의 간선을 선택할 수 있다. 그림 4.5는 그림 4.3에서 보여준 h(i)와 트리에 근거를 둔 방향그래프 G(h)를 보여준다.



[그림 4-5] 방향 그래프 G(h)

[Fig. 4-5] Directed Graph G(h)

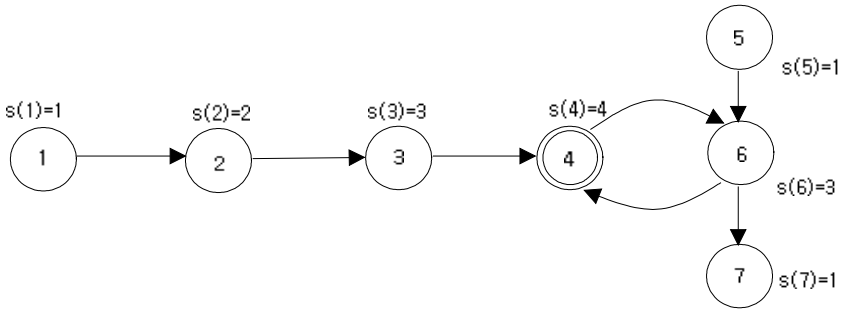
각각의 노드 i는 트리 T에서 i의 인접한 노드들 중에서 가장 큰 h(j)를 갖는 노드 j로 향하는 간

선을 갖는다. 그림 4.6은 앞의 그림 4.4에서 s(i)와 트리에 근거를 둔 방향그래프 G(s)이다. 노드 4는 가장 큰 s(j)를 갖는 인접 노드가 3, 6번 두 개이다. 이것은 가장 큰 s(j)와 가장 큰 라벨을 갖는 인접한 노드를 선택함으로서 해결된다. 따라서, 노드 4는 6번으로 향하는 간선을 갖고 3번으로 향하는 간선은 갖지 않는다.

다음의 소정리는 방향그래프 G(f)의 중요한 속성을 나타낸다.

[소정리 4.3] 방향그래프 G(f)가 정확히 하나의 사이클을 포함하고 연결되어 있다면 그 사이클의 길이는 2이다.

증명 : 집합 A(f)의 정의에 의하여 T는 연결되어 있으므로 모든 $i \in N(f)$에 대하여 $(i, j) \in A(f)$인 j의 인접한 노드 j가 정확히 하나 존재한다. j의 다른 인접한 노드 $k(k \neq j)$에 대하여 $f(k) < f(i)$가 성립한다. 순서함수의 정의에 의해 $f(k) < f(i)$ 이므로 k의 모든 인접한 노드 $k'(k' \neq j)$에 대하여 $f(k') < f(k)$라 할 수 있다. 따라서 $(k, j) \in A(f)$이다.



[그림 4-6] 방향 그래프 G(s)
[Fig. 4-6] Directed Graph G(s)

이것은 T의 모든 간선에 대하여 G(f)에는 적어도 하나의 대응되는 간선이 존재한다는 것을 의미한다. 따라서 G(f)는 연결되어 있다. 더욱이 G(f)안의 모든 노드가 정확히 하나의 간선을 갖는다는 것은 G(f)는 $|N(f)|$개의 간선을 갖는다는 것을 의미한다. G(f)가 $|N(f)|$개의 간선을 갖는 연결된 그래프이므로 이것은 정확히 하나의 사이클을 갖는다. (사이클을 이루지 않는다면 정점의 수가 간선의 수보다 하나 더 많아야 한다.)

G(f)에서 $l \geq 3$에 대하여 $(i_1, i_2, \cdots, i_l)$이 하나의 사이클을 이룬다고 가정하자. 만약 (i, j) 가 집합 A(f) 안의 간선이라면 (i, j)는 집합 E에 속하는 간선이다. 이것은 $(i_1, i_2, \cdots, i_l)$이 T 안에서 사이클을 이룬다는 것을 의미한다. 그러나 T는 사이클이 아니다. 따라서, G(f)가 포함하는 하나의 사이클은 크기가 2인 사이클이어야만 한다.

i와 j를 G(f)에 속하는 유일한 사이클인 두 개의 노드라고 하자. 일반적으로 $f(i) \geq f(j)$라고 가정

하자. G(f)에서 간선 (i, j) 와 (j, i)를 제거하면, 각각이 방향이 있는 트리가 되는 두 개의 트리를 구할 수 있다. 노드 i을 포함하는 방향 트리를 T_i(f) = (N_i(f), A_i(f))로 표현하고, 노드 j를 포함하는 방향 트리를 $T_j(f)=(N_i(f),A_j(f))$로 표현하자. 또한 i를 $T_i(f)$의 루트로 나타내고, j를 $T_j(f)$의 루트로 나타내자. $T_i(f)$와 $T_j(f)$가 새로운 루트를 갖는 트리이므로, 이와 같은 트리에서 인접한 노드의 쌍에 대하여 부모와 자식 관계의 개념을 사용할 수 있다. $T_i(f)$와 $T_j(f)$에 대한 다음과 같은 속성이 쉽게 보일 수 있다.

[정리 4.4] $T_i(f)$안에 있는 간선과 $T_j(f)$안에 있는 각각의 간선은 하나의 노드에서 그의 부모 노드로 방향이 정해져 있다. 만약 k가 $T_i(f)$ 또는 $T_j(f)$ 안에 있는 단 노드가 아니라면, k의 각각의 자 노드 l에 대하여 f(k)>f(l)이다.

위의 정리 4.4가 함축하고 있는 내용은 만약 f(i) > f(j)라면, i는 T에서 가장 큰 f(i)를 갖는다는 것을 의미하고, 반면에 f(i)=f(j)라면 i와 j가 T에서 가장 큰 f(i)를 갖는다는 것을 의미한다. 방향그래프 G(h)를 나타낸 그림 4.5에서 간선 (3, 4) 와 간선 (4, 3)을 제거함으로써 노드 3을 루트로 갖는 방향 트리 T_3(h)와 노드 4를 루트로 갖는 방향 트리 T_4(h)를 얻을 수 있다. 위의 정리 4.4는 T_3(h)와 T_4(h)에 대하여 쉽게 입증할 수 있다. h(3)=h(4)=2이므로, 정점 3과 4는 트리에 있는 모든 정점 사이에서 가장 큰 h(i)를 갖는다. 따라서 h(i)를 사용한 트리 T에서 중심을 구하는 방법은 다음과 같은 정리를 유도할 수 있다.

[정리 4.5] T 안의 모든 정점의 h(i)가 높이조건을 만족한다면, 중심(T) = { k | 모든 l에 대하여, h(k) ≥ h(l)} 이다.

증명 : h(i)가 높이조건을 만족하므로 소정리 4.1에 의하여 h는 순서함수이다. 그러므로 소정리 4.3에 의하여 G(h) = (N(h), A(h))는 길이가 2인 유일한 사이클을 갖는 연결된 그래프이다. i와 j를 G(h)에서 유일한 사이클 안에 있는 h(i) ≥ h(j)를 만족하는 두 개의 노드라고 하자. 루트를 갖는 방향 트리 T_i(h)=(N_i(h), A_i(h))와 $T_j(h)=(N_j(h),A_j(h))$를 나누어 생각해보자. 모든 k ∈ N_i(h)에 대하여 h(k)는 k를 루트로 갖는 T_i(h)의 부분트리의 높이이다. 이 사실은 다음과 같이 확인할 수 있다.

소정리 4.1은 모든 단 노드가 아닌 노드 k ∈ N_i(h)에 대하여, k의 모든 자노드에 대하여 f(k)>f(l)이 성립한다는 것을 의미한다. 그러므로 모든 단 노드가 아닌 노드 k ∈ N_i(h)에 대하여,

$N_h(k)$={h(l) | l은 $T_i(h)$에 속하는 k의 자 노드}이다.

h(i)가 높이조건을 만족하므로, 모든 단 노드가 아닌 노드 k ∈ N_i(h)에 대하여 h(k)=max

($N_h^-(k)$)+1이다. 이것을 다시 표현하면,

h(k)=max{h(l)|l은 $T_i(h)$안에 있는 k의 자 노드}+1 이다.

의 모든 단 노드의 h(i)가 0으로 주어지면, 모든 k $\in N_i$(h)에 대하여 h(k)는 k를 루트로 갖는 T_i(h)의 부분 트리의 높이라는 것을 귀납적으로 알 수 있다. h(i)와 h(j)의 관계에 근거를 둔 두 가지 부분적인 경우에 대하여 살펴보자.

1) h(i)>h(j)인 경우

이 경우는 정리 4.4로부터 i가 T에서 가장 큰 h(i)를 갖는 정점이라는 것을 알 수 있다. 그러므로, max{k | h(k) $\geq$ h(l) for all l $\in$V} = {i} 이다. 여기서, 중심(T) = {i}임을 보이자.

h가 순서함수이고, (i, j) $\in$A(h)이므로 j가 i의 모든 인접한 노드들 중에서 가장 큰 h(i)를 갖는다. l을 T_i(h)에서 가장 큰 h(i)를 갖는 i의 자 노드라고 하자. 그러면 h(j) $\geq$ h(l) 이다. 더욱이 h(i)>h(j)와 h(i)=h(l)+1이 성립한다. 따라서 h(l)=h(j)이며, h(i)=h(j)+1이다.

T_i(h)의 높이가 루트 i와 T_i(h)에서 i와 가장 멀리 떨어진 노드 사이의 거리이므로, e(i)=max{h(i), h(j)+1}=h(i)의 관계식을 얻을 수 있다. 같은 방법으로 e(j)=max{h(i)+1, h(j)}=h(i)+1이다.

N_i(h) - {i}에 속하는 모든 노드들은 T_i(h)에 속하는 가장 먼 노드로부터 적어도 h(j)+2=h(i)+1의 거리를 갖는다. 그리고 N_j(h) - {j}에 속하는 모든 노드는 T_i(h)에 속하는 가장 먼 노드로부터 적어도 h(i)+2의 거리에 있다. 이것은 i가 가장 작은 이심률을 갖는 T안의 정점이라는 것을 의미한다. 따라서, 중심(T)={i}이며, h(i)>h(j)인 모든 경우에 대하여 정리는 참이다.

2) h(i)=h(j)인 경우

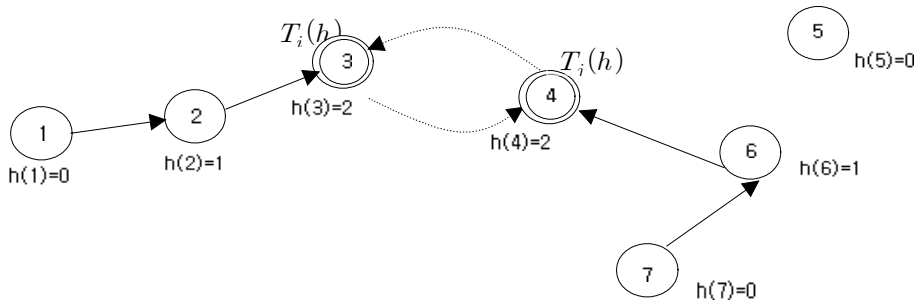
이 경우에 있어서, 정리 4.4는 i와 j가 가장 큰 h(i)를 갖는 T안의 두 정점이라는 것을 의미한다.

따라서, {k | h(k) $\geq$ h(l) for all l $\in$V}={i, j}이다. e(i)=e(j)=h(i)+1=h(j)+1 임은 자명하다.

첫 번째 경우에서 N_i(h) - {i} 안의 모든 노드들은 T_j(h)안에 있는 가장 먼 노드로부터 적어도 거리가 h(j)+2 만큼 떨어져있고, N_j(h) - {j}안의 모든 노드들은 T_i(h)안의 가장 먼 노드로부터 적어도 거리가 h(i)+2만큼 떨어져 있다. 그러므로 i와 j는 가장 작은 이심률을 갖는 T의 두 정점이다. 그러므로 중심(T)={i,j}이다. 따라서 $h(i)=h(j)$일 때도 정리는 참이다.

위의 정리에 관한 두 가지 경우의 증명은 순서적이다.

우선 소정리 4.1과 정리 4.5의 모든 $h(i)$가 높이조건을 만족한다면 각각의 프로세스는 자신이 T의 중심인지 아닌지를 단순히 인접한 노드의 $h(i)$를 살펴봄으로써 결정할 수 있다. 특히, 프로세스가 자신의 $h(i)$가 모든 인접한 노드들의 $h(i)$보다 크거나 같다는 사실을 발견할 수 있다면, 그 자신이 T의 중심이라고 정의할 수 있다.



[그림 4-7] h(i)=h(j)인 경우

[Fig. 4-7] Case in h(i)=h(j)

[정리 4.6] T 안의 모든 정점의 $s(l)$가 크기조건을 만족한다면,
중앙 (T) = { k | $s(k) \geq s(l)$ for *all* $l \in V$} 이다.

증명 : i와 j를 G(s)안의 유일한 사이클 안에 놓인 s(i) ≥ s(j) 을 만족하는 두 개의 노드라고 하자. 모든 노드 k ∈ N_i(s)에 대하여, s(k)는 k를 루트로 갖는 $T_i(s)$의 부분트리의 크기이다. 같은 방법으로, 모든 노드 k ∈ N_j(s)에 대하여, s(k)는 k를 루트로 갖는 의 부분트리의 크기이다. s(i)와 s(j) 사이의 관계에 근거를 둔 두 가지의 경우를 살펴보자.

1) s(i)>s(j) 인 경우

이 경우, 소정리 4.2는 i가 가장 큰 s(i)를 갖는 T의 정점이라는 사실을 나타낸다. 따라서 {k | s(k) ≥ s(l) for all l ∈V} = {i}이다.

중앙(T) = {i}임을 보이자.

웨이트의 정의에 의하여, w(j)=w(i) + s(i) - s(j)라는 식을 얻을 수 있다. s(i)>s(j)이므로 w(i)<w(j)이다. 다음에 모든 k ∈ N_i(s) -{i}에 대하여 w(i)<w(k) 임을 보이자.

$T_i(s)$의 자노드 k를 고려하자. 웨이트의 정의에 의하여

w(k)=w(i)+(|V|-s(k))-s(k) 라는 식을 얻을 수 있다. |V|-s(k) ≥ s(j)+1이고 s(j) ≥ s(k) 이므로, |V|-s(k) > s(k)이고 따라서 w(i)<w(k)이다.

따라서 모든 노드 k ∈ N_i(s) -{i}에 대하여 w(i)<w(k)가 성립한다는 사실을 유도할 수 있다. 같은 방법으로, 모든 노드 k ∈ N_j(s) -{j}에 대하여 w(k) > w(j)이다. 이것은 i가 가장 작은 웨이트를 갖는 T 안의 정점이라는 것을 의미하며 따라서, 중앙(T)={i} 이다.

2) s(i) = s(j) 인 경우

이 경우에 있어서, 소정리 4.2는 i와 j가 가장 큰 s의 함수값을 갖는 T안의 두 정점이라는 사실을 나타낸다. 따라서 {k | s(k) ≥ s(l) for all l ∈V} = {i, j} 이다.

다음에 중앙(T)={i, j} 임을 보이자. 1)의 경우와 같은 방법으로, 모든 k ∈ N_i(s) -{i}에 대하여, w(k) > w(i)이고, 모든 k ∈ N_j(s) -{j}에 대하여, w(k) > w(j)이다. 정점의 값의 정의에 의하여,

w(j)=w(i) + s(i) - s(j) 라는 식을 얻을 수 있다. s(i)=s(j) 이므로 또한 w(i)=w(j) 이다. 그러므로 i와 j는 가장 작은 값을 갖는 T안의 두 정점이다. 따라서 중앙(T) = {i}이다.

위의 정리 4.6에 관한 두 가지 경우의 증명은 순서적이다. 우선 소정리 4.2와 정리 4.6의 모든 s(i)가 크기조건을 만족한다면 각각의 프로세스는 자신이 T의 중앙인지 아닌지를 단순히 인접한 노드들의 s(i)를 살펴봄으로써 결정할 수 있다. 특히 프로세스가 자신의 s(i)가 모든 인접한 노드들의 s(i)보다 크거나 같다는 것을 발견했을 때 자신이 T의 중앙이라고 정의할 수 있다.

4.3 알고리즘의 유한성

중앙을 찾는 알고리즘이 유한한 횟수의 변화 후에 종료된다는 것을 증명하는 것은 중심을 찾는 알고리즘과 거의 동일하므로 생략하고 중심을 찾는 알고리즘의 종료에 대하여 증명한다.

만약 프로세스 i의 변화에 의하여 프로세스 i의 h(i)가 감소한다면 그 변화를 "감소변화" 라고 하고 그렇지 않으면 "증가변화"라고 하자. 중심을 찾는 알고리즘의 실행 열은 모든 프로세스에 있는 조건이 거짓이 되는 순열의 마지막 변화를 갖는 유한한 연속적인 변화의 순열이라고 정의하자. 여기서 증명하고자 하는 것은 중심을 찾는 알고리즘의 모든 가능한 실행 열이 프로세스의 초기 h(i)에 관계없이 유한하다는 것이다. 중심을 찾는 알고리즘의 임의의 실행 열을 고려해보자.

M(i, p)를 실행 열안에 있는 프로세스 i에 의한 p번째 변화라고 하자. 또한 M(i, p, ↓)를 감소하는 변화라고 하고, M(i, p, ↑)를 증가하는 변화라고 하자. 알고리즘에 의하여, 프로세스 i에 의하여 실행된 변화의 영향을 받아 실행되는 변화 열은 M(i, p),M(i+1, p+1),M(i+2, p+2),… 으로 나타낼 수 있다. T 안의 모든 프로세스는 이와 같은 실행 열 안에 속하게 되는데, 이 실행 열을 연속적으로 h(i)가 감소하는 변화 열과 h(i)가 증가하는 변화 열로 나누어 생각하자.

4.3.1 감소하는 변화

부분 실행 열은 T안의 프로세스가 n개이므로 최대 n번의 변화들로 이루어 질 수 있다. 프로세스 i에 의하여 시작된 변화 열은 프로세스 i의 h(i)가 변경되기 전에는 안정된 상태로 변화 없이 남아 있게 된다. 임의의 i에 의한 감소변화 열 M(i, p),M(i+1, p+1),M(i+2, p+2),… 에 대하여, 만약 t=i-1인 프로세스 t의 변화 M(t, p')(감소하는 변화일 수도, 증가하는 변화일 수도 있다.) 의하여 프로세스 i가 감소변화를 하게 된다면 t에 의한 감소변화의 수는 i에 의한 감소변화의 수보다 1이 더 많다. 이것은 높이 함수 h(i)가 순서함수이므로 당연한 것이다. T 안의 프로세스의 수는 유한하므로, 프로세스의 총수를 n이라고 할 때, 프로세스 i에 의하여 시작된 감소변화 열은 최소 한번부터 최대 n번의 감소하는 변화를 할 수 있다. 각각의 프로세스 i의 고유번호가 $1 \le h(i) \le n$ 이므로, 프로세스 0은 최대 n번의 변화를 포함하는 감소변화 열을 이룰 수 있다. 분산된 환경에서는 각각의 프로세스들이 동시에 다른 프로세스와 무관하게 알고리즘을 수행 하므로 감소변화 열을 이루는 프로세스의 실행 순서는 무작위가 될 수도 있다. 그러나 각각의 프로세스의 이접한 노드들을 고려

할 때 연속적으로 구성된 감소변화 열을 만들 수 있다. 이것은 각각의 프로세스들이 개별적으로 알고리즘을 수행하더라도, 인접한 노드의 값을 확인하여 자신의 값을 바꾸게 되므로 수행의 순서에 관계없이 단지 인접한 노드의 변화에만 영향을 받는다는 이유 때문에 가능한 것이다. 또한, 이러한 감소변화 열을 이루는 변화의 주체인 프로세스는 안정된 상태에 있으므로 선두의 변화를 실행한 프로세스의 상태가 변하기 전에는 그들의 상태를 바꾸지 않는다. 따라서 모든 감소변화의 실행 열에 포함된 변화의 총수는 유한하다.

[소정리 4.7] 중심을 찾는 알고리즘에 의한 감소변화의 열은 유한하다.

4.3.2 증가하는 변화

중심을 찾는 알고리즘의 임의의 실행 열이 있다고 가정할 때, 알고리즘이 만드는 감소변화의 수는 유한하므로 알고리즘에 의하여 만들어지는 모든 감소변화를 포함하는 실행열의 유한한 접두어가 존재한다. 그와 같은 접두어의 길이를 t라고 하자. 그러면 실행 열 E=M(_, t+1),M(_, t+2),M(_, t+3),… 은 오직 증가변화만을 포함한다. 여기서의 목표는 E가 유일한 열이라는 사실을 밝히는 것이다. E가 무한하다고 가정하여 모순을 유도하자. E 안의 무한히 많은 변화를 만드는 프로세스 j_0가 적어도 하나 존재한다고 가정하자. 단노드는 많아야 한 번의 변화를 만들 수 있다. 따라서 j_0는 단노드가 아니고, 적어도 두 개의 인접한 노드들을 갖는다. 분명히 j_0는 적어도 하나의 단노드가 아닌 인접한 노드를 갖는다. 이를 j_1이라고 할 때, j_1은 E안에서 무한히 많은 변화를 한다. 이제 j_0가 E안에서 무한히 많은 변화를 만드는 인접한 노드들을 적어도 2개 이상 갖는다는 사실을 증명하자. j_1이 E 안에서 무한히 많은 변화를 만드는 j_0의 유일한 인접한 노드라고 가정하자. 그러면 j_0의 인접한 노드가 아닌, t' > t가 존재하여 다음과 같은 실행 열을 만든다.

E'=M(_, t'+1),M(_, t'+2),M(_, t'+3),…

j_0와 j_1이 E' 안에서 무한히 많은 변화를 만든다. 분명히, E'안에서 유한하게 많은 변화를 실행한 후에, h(j_1)은 j_0의 다른 모든 인접한 노드들의 h(i)보다 크다. 이것은 E' 안의 유한하게 많은 변화가 실행된 후에 $N_h^-(j_0)$가 더 이상 h(j_1)을 포함하지 않는다는 것을 의미하며, 따라서 부분열이 변하지 않고 남아있게 된다. 만약 $N_h^-(j_0)$가 변하지 않는다면, j_0가 더 이상 변화를 만들 이유가 없다. 이것이 내포하는 것은 j_0는 E' 안에서 유한하게 많은 변화를 만든다는 것이다. 이것은 j_0가 E안에서 무한히 많은 변화를 만든다는 가정에 모순된다. 그러므로 j_0는 E안에서 무한히 많은 변화를 만드는 단 노드가 아닌 인접한 노드들을 적어도 2개 이상 갖는다. 마찬가지로 j_1에서도 이러한 사실이 적용된다. 그러므로 j_1은 단 노드가 아닌 j_0와 같지 않은 무한히 많은 변화를 만드는 인접한 노드 j_2을 갖는다. 이것은 E안에 무한히 많은 변화를 만드는 단 노드가 아닌 서로 다른 프로세스의 무한한 열이 존재한다는 의미이다. 그러나 이것은 T가 유한한 프로세스를 갖는다는 사실

에 모순이다. 따라서 초기에 가정한 E가 무한한 실행열을 갖는다는 사실은 거짓이 된다. 위의 결과를 요약하면 다음과 같다.

[소정리 4.8] 중심을 찾는 알고리즘에 의한 증가변화의 수는 유한하다.
소정리 4.7과 소정리 4.8에 따라서 다음 정리를 유도 할 수 있다.

[정리 4.9] 중심을 찾는 알고리즘에 의해 수행되는 변화의 총 수는 유한하다.
이것은 중심을 찾는 알고리즘이 유한한 수의 변화를 한 후에 종료된다는 것을 의미한다. 알고리즘이 종료되었을 때, 가장 큰 h(i)를 갖는 정점이 T의 중심이다.

5. 결론

오늘날과 같이 분산시스템이 고도로 발달된 시대는 자율안정 알고리즘은 큰 의미를 가진다. 자율안정 알고리즘의 응용으로서의 분산 환경이라는 특수성 때문에 발생하는 문제점을 해결하는 것이 중요한 해결 과제임을 알 수 있다. 이를 위해 결함 제한을 접목시키는 연구가 진행되었으나, 결함제한을 접목시키는 방법도 역시 어려운 문제로 남아있다. 많은 시스템이 연결되어있는 분산 환경에서 구성원인 어느 시스템이 오류를 일으키게 될 때, 시스템이 외부의 어떠한 간섭 없이 자신과 이웃의 정보만을 이용하여 스스로 그 오류를 교정하고 안정된 상태로 복귀할 수 있다는 사실은 오늘날의 거대해진 분산 시스템에서 꼭 필요한 특성이다. 이 논문에서는, 분산 시스템에서 중심과 중앙을 찾기 위하여 시스템을 구성하는 각각의 프로세스가 다른 프로세스와 비동기적으로 동일한 알고리즘을 수행하였다. 또한 자신과 이웃의 정보만을 이용하여 위치를 파악하고 중심과 중앙을 발견하였다. 자율안정 알고리즘의 한계점에서 드러난 것처럼 안정된 상태에 도달하는 동안 정상인 프로세스도 자신의 상태를 변화시키는 바람직하지 못한 움직임을 보이므로, 결함 제한과 같은 특성을 접목시켜 중심과 중앙을 찾는 알고리즘을 수행하는 동안 불필요한 동작을 제한한다거나 한 프로세스에서 발생한 오류로 인한 영향이 다른 프로세스에 미치는 범위를 최소로 할 수 있는 알고리즘의 개발이 필요하다. 앞으로의 연구 과제는 자율안정의 경비에 대한 연구 및 자율안정과 결함허용의 결합에 관한 연구 등 보다 폭넓은 연구 및 알고리즘의 개발이 필요하다.

참고 문헌

[1] L. Lamport, "Unsolved problems, Solved problems, and non-problems in concurrency", Invited address, 3rd Annual ACM Symposium on Principles of Distributed Computing, 1984.

[2] E. W. Dijkstra, "Self-Stabilizing Systems in spite of Distributed Control", ACM Vol. 17, No. 11, pp. 643-644, Nov. 1974.

[3] M. Schneider, "Self-Stabilization", ACM Computing Surveys, 1993.

[4] Z. Shi and P. K. Srimani, Handbook on Theoretical and Algorithmic Aspects of Sensor/ Ad-Hoc Wireless and Peer-to-Peer Networks, Auerbach Publications, pp.393-402, 2005.

[5] M. Flatebo, A. K. Datta, S. Ghosh, "Self-Stabilization in Distributed Systems", 1995.

[6] W. Goddard, S.T. Hedetniemi and Z. Shi, “An anonymous self-stabilizing algorithm for 1-maximal matching in trees”, The Internat. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'06), Las Vegas, USA, 2006.

[7] S. T. Hedetniemi, D. P. Jacobs and P. K. Srimani, “Fault Tolerant Distributed Coloring Algorithms That Stabilize in Linear Time”, IEEE IPDPS-2002 Workshop on Advances in Parallel and Distributed Computational Models, Orlando, USA, April 2002.

저자 소개

김장환 (Jang-Hwan Kim)

1980년 서울대학교 경제학학사
1997년 한국과학기술원 전산학 석사
2003년 충북대학교 전산학박사
1984년~1988년 쌍용정보통신 연구원
1988년~1993년 Qnix Data System 연구원
1993년~1998년 SK Telecom 중앙연구원 연구원
1998년~2005년 대덕대 교수
2005년~현재 성결대 공대 교수
관심분야 : Information Security, Mobile&Wireless Communication, Performance Analysis of Networks, Database System, Mobile Multimedia, Mobility Managements, Mobile Embedded System, Ubiquitous Computing, 알고리즘 및 계산이론, 결함허용, 정보통신 경제 예측

이충세 (Chung-Sei Rhee)

1979년 Univ. of South Carolina 컴퓨터과학과 석사
1990년 Univ. of South Carolina 컴퓨터 과학과 박사
Univ. of North Dakota 전산학과 조교수
1991년~현재 충북대학교 전자정보대학 컴퓨터공학부 교수
관심분야 : 결함허용, 알고리즘, 전문가시스템, 정보보안