Chapter 8. Graph theory

Graph model: 정점(vertex, node)과 간선(가지, edge, branch)의 결합으로, 객체(object) 상호간의 일대일 관계(pairwise relations)를 modeling 하는데 사용되는 수학적 구조.

2010 년 world cub 스페인과 네덜란드의 결승전 승패를 예측하기 위하여 London 소재 Queen Mary 대학 수학자들은 graph 이론을 사용하였다. 이들은 출전한 모든 team 에 대해 team 내 두 선수 사이에 공 pass 횟수가 얼마나 되는지 통계치를 수집했다. 이를 바탕으로 team 에서 공을 가장 많이 주고받는 두 선수가 누구인지 보여주는 pass network 를 만들었다. 그리고 축구팀의 특성이 축구공 pass network 에 나타남을 보였다.

한 경기를 치르는 동안, 모든 관심이 집중되는 선수는 득점을 하는 공격수이지만, pass network 는 득점에 중요한 역할을 한 선수가 누군지를 보여 주었다. 수학자들은 pass network 를 이용해 경기장에서 공이 어느 방향으로 진행되는지 쉽게 파악할 수 있었다. 경기결과를 예측할 수 있는 흥미로운 지표가 나왔고, 결과 결승전의 승자를 맞출 수 있었다.

Applications

(a) 고속도로 상태조사

미국 Montana 주에 있는 도시를 연결하는 고속도로가 있다. 한 도시에서 출발하여 모든 고속도로의 상태를 점검한 후, 출발 도시로 돌아온다.

Node: 도시
Branch: 두 도시를 연결하는 고속도로. 한 쌍의 도시를 연결하는 고속도로가 없을 수도 있다.
문제정의: Node-Gre 에서 출발하여, 모든 branch 를 한번만 지나면서 node-Gre 로 돌아오는 폐경로(closed path)가 존재하는가?

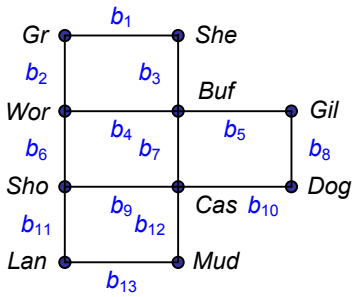


Fig. 8.1 고속도로 상태조사 – 예

Answer: NO
Consider, for example node-Wor. It is connected to 3 neighboring nodes. Two of them correspond to entering and exiting path. We cannot visit the same node without passing through the remaining path twice. 해가 존재하려면 모든 node 가 짝수개의 branch 와 연결되어야 한다.

(b) Robot 경로 최적화

자동화 robot 을 사용하여 PCB 기판에 5 개의 구멍을 뚫고자 한다. 구멍의 위치가 고정되어 있을 때, 5 개의 구멍을 가장 효율적으로 뚫는 robot 의 경로를 찾는다.

Node: 구멍
Branch: 각 구멍을 잇는 가상의 선. 모든 node 쌍은 branch 로 연결되며, 각 branch 는 측정

가능한 길이를 갖는다. 이를 가중치(weight)라 한다.
문제정의: 경로의 길이를 경로상에 있는 모든 branch weight 의 합으로 정의하고, 모든 node 를 한번씩만 경유하는 최소길이의 폐경로를 찾는다.

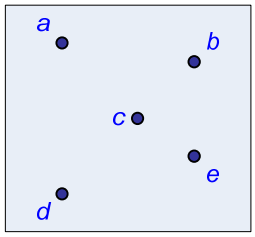


Fig. 8.2 Soldering machine: Example

(c) Traveling salesman problem

A 회사는 10 개의 도시에 지점을 갖고 있고, B 는 지점이 있는 도시에 살고 있다. B 는 매주 각 지점을 방문하여 업무를 처리하여야 한다.

Node: 도시의 각 지점
Branch: 지점이 위치한 도시를 연결하는 도로
문제정의: 모든 경로를 한번씩만 지나는 최소한의 여행거리를 갖는 폐경로를 찾는다.

(d) Bacon number in sociology

The Bacon number of an actor or actress is the number of degrees of separation he or she has from Bacon. The higher the Bacon number, the farther away from Bacon the actor is. The computation of a Bacon number for a actor X is a “shortest path” algorithm:

- Bacon himself has a Bacon number of 0.
- Those actors who have worked directly with Bacon have a Bacon number of 1.
- X와 공연한 배우 중에서 Bacon number 가 가장 작은 사람의 Bacon number 가 N이면, X의 Bacon number 는 $N+1$이다.
- 예를 들어, Elvis Presley 가 Edward Asner 와 영화 ‘Change of habit(1969)’에 함께 출연하였고, Edward Asner 는 Bacon 과 ‘JFK(1991)’에 공동 출연하였으면, Asner 와 Presley 의 Bacon number 는 각각 1과 2이다.
- Hollywood 영화산업에 종사하는 사람은 Bacon number 6 이내로 추정되고 있다.

(e) Similarity (유사도) graph

Program 숙제를 냈는데 일부 학생이 다른 학생의 program 을 copy 하여 제출하였다. 불량학생을 색출하기 위하여, program 간의 유사성을 판별하고자 한다.

판별 기준: 예를 들어,
- c1: program 의 statement 수
- c2: program 에서 특별 명령문(statement, e.g. return) 수
- c3: 특정 함수 호출 회수

	c1	c2	c3
p1	66	20	1
p2	41	10	2
p3	68	5	8
p4	90	34	5
p5	75	12	14

Node: program 각각을 (p_1, p_2, p_3)로 표시하되, 각 성분은 가중치로 표시: *e.g.* program-1 의 경우, 판별기준을 사용하여 $(66, 20, 1)$로 표시

Branch: Given a node-pair $v = (p_1, p_2, p_3)$와 $w = (q_1, q_2, q_3)$, define a similarity measure

$s(v, w) = |p_1 - q_1| + |p_2 - q_2| + |p_3 - q_3|$

e.g. $s(p_1, p_3) = |66 - 68| + |20 - 5| + |1 - 8| = 24$: Small s implies similar programs.

문제정의: 미리 설정된 문턱값 S를 사용하여, $s(v, w) < S$이면, node-pair (v, w) 사이에 branch 를 연결한다. We say a node-pair (v, w) is in the same class, if $v = w$ or they are connected.

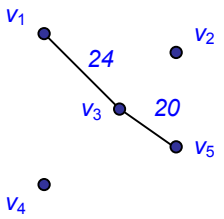


Fig. 8.3 $S = 25$로 설정된 경우의 class 구성

(f) Hypercube (n-cube)

Some algorithm could be executed efficiently using parallel (병렬) computing compared to serial computing.

- 많은 graphic card 는 병렬 computing 작업을 수행하기 위한 구조로 구성되어 있다. 이는 많은 영상처리 기법들이 병렬연산으로 효율적인 연산이 가능하기 때문이다 (*e.g.* fractals). 현재 AI 역시, 병렬연산을 많이 사용하기 때문에, AI 로 인한 최대 수혜주로써 graphiic card 제조회사가 대표적인 이유인 것이다.
- 병렬 algorithm *vs*. cloud computing
- n-cube 는 2^n개의 processor 로 구성된다. 그리고 각각의 processor 는 서로 통신이 가능하도록 연결되어 있다.

Node: 각 processor. Processor 각각은 local memory 를 갖는다.

Branch: A branch connects a node-pair, if binary representation of node number differs only 1-bit.
e.g. 3-cube 의 경우, node (110)은 정확하게 3개의 node 와 연결된다. $\{010, 100, 111\}$

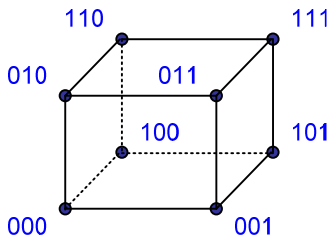


Fig. 8.4 3-cube 의 구성

문제정의: 두 개의 $(n-1)$-cube 를 연결하여 n-cube 를 구성하자.
먼저 n-cube 가 갖고 있는 branch 의 수를 살펴보자.

1-cube: node $:= \{0,1\}$, 1-branch

2-cube: node $:= \{00, 01, 10, 11\}$, Each node has 2 branches. Since direction of branch does not

matter, there are $4(\text{nodes}) \times 2(\text{branchs/node})/\,2$, which is 4-branches.
n-cube: $2^n\ (\text{nodes}) \times n(\text{branchs per node})/\,2 = n \cdot 2^{n-1}$ branches

연결방법: 두 개의 $(n-1)$-cube 를 각각 H_1과 H_2라 하자. 이진표현 된 H_1의 각 node 번호 앞에 '0'을, H_2의 각 node 번호 앞에 '1'을 추가하고, (1-bit 추가 전) 같은 번호를 갖는 H_1과 H_2의 node-pair 만을 서로 연결한다. (H_1과 H_2의 node 는 이미 첫 번째 bit 를 달리하기 때문에 다른 node 와는 연결될 수 없다). 따라서, 2^{n-1} 개의 branch 가 추가된다. 따라서, 각 $(n-1)$-cube 에서의 branch 총 수는 $2((n-1)2^{n-2}) + 2^{n-1}$ 이다.

Definitions

(a) *Un-directed graph*: A (un-directed) graph G consists of a set N of nodes (or vertices) and a set B of branches (or edges) such that each branch $b \in B$ is associated with an un-ordered pair of nodes. (v,w) denotes a branch between nodes v and w. We say that a branch b is *incident* on v and w.

(b) *Parallel branches* (병렬 가지)

(c) *Loop*

(d) *Isolated node* (고립 정점)

(e) *Simple graph* (단순 그래프)

(f) *Complete graph*, K_n: The complete graph on n-nodes, denoted K_n, is the simple graph with n-nodes in which there is a branch between every pair of distinct nodes.

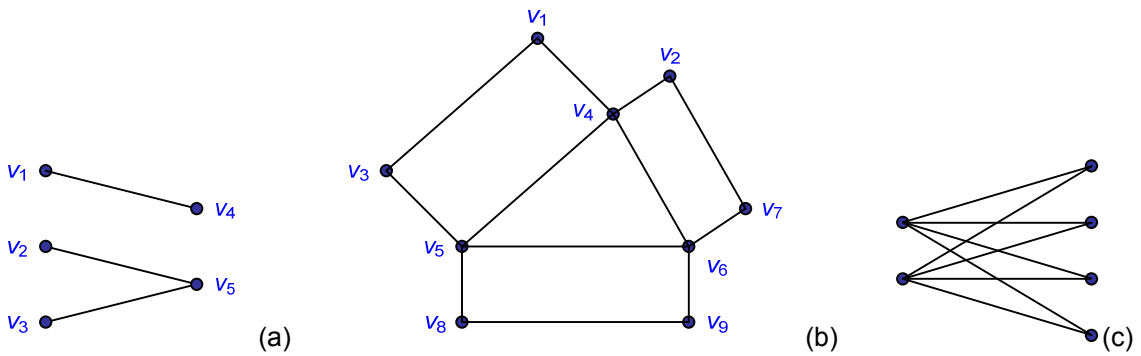
(g) *Bipartite* (이분): A graph is said to be bipartite, if $\{N_1, N_2\}$ is a partition of N and each branch in B, if exists, connects one node in N_1 and the other in N_2.

(h) *Complete bipartite graph* on m and n nodes, $K_{m,n}$: A bipartite graph with

① $N = N_1 \cup N_2$ with $N_1 \cap N_2 = \emptyset$ (*i.e.* partition of N), $|N_1| = m$, and $|N_2| = n$,
② All nodes in N_1 are connected to every nodes in N_2.

Party 에 참석하였다. 참석자들이 모두 아는 사이일 수도 있고, 아무도 알지 못하는 어색한 party 일 수도 있다. 물론 아는 사람과 알지 못하는 사람이 섞여 있을 확률이 더 높다. 그래프 이론을 이용하면 party 에 온 사람 중에 누가 서로 아는 사이인지 확실하게 알 수 있다. 두 사람 사이의 관계를 선으로 나타내고, 이미 친구인 예를 들어, 6명이 모여 있을 경우, 서로 아는 사람이 3명 있거나 서로 모르는 사람이 3명 있을 것이다.

Example 8.1



(a) Choose $V_1 = \{v_1, v_2, v_3\}$, and $V_2 = \{v_4, v_5\}$: Bipartite graph

In a bipartite graph, if a branch b connects a node-pair (v, w), then v and w must belong to different subset of nodes.

(b) Not a bipartite graph:

Disproof by contradiction: Bipartite graph 라 가정하자. 그러면, 9 개의 node 로 구성된 node 집합 N을 N_1과 N_2의 partition 으로 구분할 수 있다.

Consider nodes v_4, v_5, and v_6. Since v_4 and v_5 are connected, they must belong to different subset. Let $v_4 \in N_1$. Then, $v_5 \in N_2$. Likewise, the connection between v_5 and v_6 implies that $v_6 \in N_1$. Thus, both v_4 and v_6 belong to the same subset, which contradicts the connection between them.

8.2 Path and cycle

Definitions Let $G = (N, B)$ be a graph and v and w be nodes in G.

(a) *Path* (경로): Let n_0 and n_k be nodes in a graph. A path from n_0 to n_k of length-k is an alternating sequence of $(k+1)$-nodes and k-branches beginning with node n_0 and ending at node n_k, $(n_0, b_1, n_1, b_2, \ldots, n_{k-1}, b_k, n_k)$.

(b) *Connected graph* (연결된 그래프): A graph G is connected, if, for any node-pair (v, w), there is a path from v to w.

(c) *Subgraph*: We say (N', B') a subgraph of G, if

① $N' \subseteq N$ and $B' \subseteq B$

② For every branch $b' \in B'$, if $b' = (v', w')$, then $v', w' \in N'$.

(d) *Component*: The subgraph G' of G consisting of all branches and nodes in G that are contained in some path beginning at v is called the component of G containing v.

(e) A *simple path* from v to w is a path from v to w without repeated nodes.

(f) A *cycle* is a path of non-zero length from v to v without repeated branches.

(g) A *simple cycle* is a cycle from v to v in which, except for the beginning and ending nodes that are both equal to v, there are no repeated nodes.

(h) *Degree of a node-v*, denoted as $\delta(v)$ is the number of branches connected to v: Each loop on v contributes 2 to the degree of v.

Example 8.2

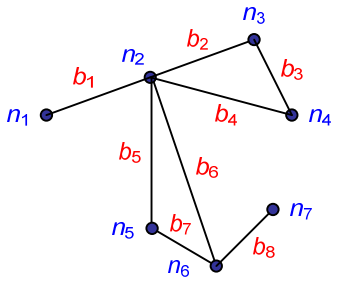


Fig. 8.5

(a) Path of length-4 from node-n_1 to node-n_2: $(n_1, b_1, n_2, b_2, n_3, b_3, n_4, b_4, n_2)$

In a simple graph, a node-pair is connected by a single (unique) branch. Thus, the path can be denoted by only nodes, say $(n_1, n_2, n_3, n_4, n_2)$.

(b) Consider the following paths:

$(6, 5, 2, 4, 3, 2, 1)$

$(6, 5, 2, 4)$

$(2, 6, 5, 2, 4, 3, 2)$

$(5, 6, 2, 5)$

(7)

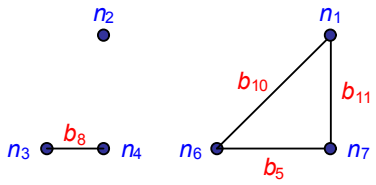


Fig. 8.6

(b) Subgraph

A subgraph may consist of only one node, one node-pair with a connecting branch, so on.

(c) Konigsberg 7 bridge problem

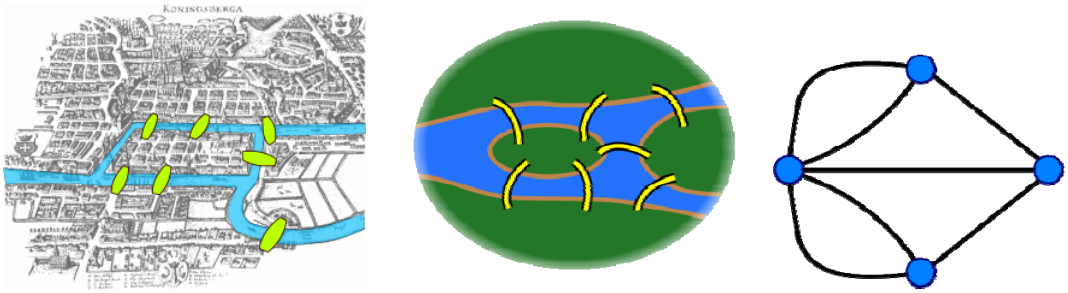


Fig. 8.7 Konigsberg 7 bridge and its graph model

Problem is to start at any location, $A \sim D$, walk over each bridge exactly once, and then return to the starting position. In graph model, nodes represent the locations and the branches the bridges. Thus, we want to find a cycle in the graph that includes all the nodes and all of the branches.

Euler proved that the problem has no solution (node-A에 연결된 branch 의 수가 홀수). In honor of him, a cycle in a graph that includes all the nodes and all of the branches is called an *Euler cycle*.

Theorem 8.1 If a graph G has an Euler cycle, then G is connected and every nodes has even degree.

Theorem 8.2 If G is a connected graph and every node has even degree, then G has an Euler cycle.

Theorem 8.3 If G is a graph with m-branches and k-nodes, $N = \{n_1, n_2, \ldots, n_k\}$, then

$\sum_{i=1}^{k} \delta(n_i) = 2m$

That is, the sum of degrees of all nodes in a graph is even.

← In any graph, the number of nodes with odd degree is even.

Theorem 8.4 A graph has a path with no repeated branches from v to w contains all the nodes and branches, if and only if it is connected and v and w are the only nodes having odd degrees.

연결된 그래프에서, 한 쌍의 node, v와 w, 만 홀수 차수를 갖고, 나머지는 짝수 차수를 갖는다고 하자. v와 w를 branch-e로 연결하고 새로운 그래프 G'을 구성하면, G'은 Euler cycle 을 갖는다. v로 시작하는 Euler cycle 에서 branch-e를 제거하면, v에서 w로 가는 모든 node 와 branch 를 반복 없이 경유하는 경로를 구할 수 있다.

Theorem 8.5 If a graph G contains a cycle from v to v, G contains a simple cycle from v to v.

8.3 Hamiltonian cycles and traveling salesman problem

Problem statement (*Hamiltonian puzzle*): Start at any city, traveling along the branches, visit every city (node) exactly once, and return to the initial city.

Hamiltonian puzzle 은 모든 node 들을 정확히 한번 지나는 cycle 을 발견하면 풀린다. 이러한 특성을 갖는 cycle 을 *Hamiltonian cycle* 이라 부른다.

Example 8.3

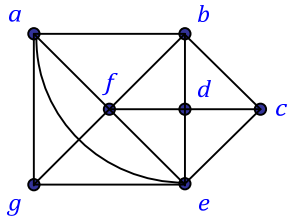


Fig. 8.8

(a) Hamiltonian cycle 과 Euler cycle 을 찾는 문제는 일견 비슷해 보이지만, Euler cycle 은 각 branch 를 한번씩 지나는 반면 Hamiltonian cycle 은 각 node 를 한번씩 지나는 점에서 차이가 있다. Fig. 8.8 의 graph 는 홀수 차수를 갖는 node 가 있기 때문에 Euler cycle 이 존재하지 않는다. 반면에 Hamiltonian cycle 은 존재한다. The cycle (a, b, c, d, e, f, g, a) is a Hamiltonian cycle.

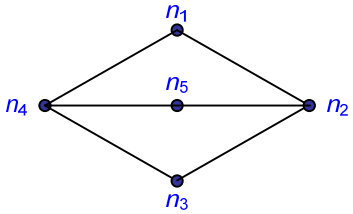


Fig. 8.9

(b) A Hamiltonian cycle, if exists, would be like $(n_a, b_1, n_b, b_2, n_c, b_3, n_d, b_4, n_e, b_5, n_a)$. Graph 에 5개의 node 가 있기 때문에, Hamiltonian cycle 은 정확히 5개의 branch 만을 포함할 있다. 그림에는 6개의 branch 가 있으므로, branch 중 불필요한 하나를 제거하여 Hamiltonian cycle 을 구성할 수 있다고 가정하자. n_2와 n_4 node 를 제외한 모든 node 가 degree-2이기 때문에, n_2와 n_4 node 와 연결된 branch 를 하나씩 제거해야 하는데, 그러면 4개의 branch 만을 남기게 되어, Hamiltonian

cycle 을 구성할 수 없게 된다.

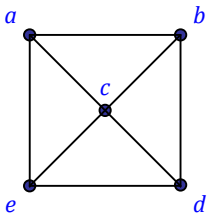


Fig. 8.10 A graph with a Hamiltonian cycle

위의 논리를 Fig. 8.10 의 graph 에 적용해 보자. 5개의 node 가 있으므로, Hamiltonian cycle 은 5개의 branch 를 포함해야 한다. Node-c와 연결된 두 개의 branch, node-$\{a, b, d, e\}$와 연결된 하나의 branch 를 제거한다고 하자. 원래 8개의 branch 중에서 6개의 branch 를 제거하면 2개의 branch 만이 남는다. 따라서, Hamiltonian cycle 은 존재하지 않는다.

이 논리는 잘못된 것이다. 경로 (a, b, c, d, e, a)가 Hamiltonian cycle 이기 때문이다. 논리가 잘못된 이유는 branch 의 삭제 과정에서 이미 제거된 branch 를 고려하지 않았다는 점이다. 예를 들어, node-c에 연결된 branch 를 삭제하면, 나머지 node 들은 2차가 되어 더 이상 연결된 branch 를 제거하지 않아야 한다.

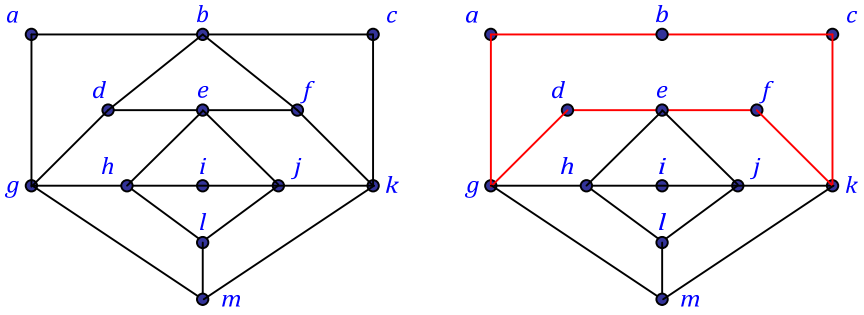


Fig. 8.11

(c) Hamiltonian cycle, H가 존재한다고 가정하자.

① Node $\{a, c\}$는 2차이기 때문에 이와 연결된 branch, (a, b), (a, g), (b, c), (c, k)는 반드시 H에 포함되어야 한다.

② Branch (b, d), (b, f)는 H에 포함될 수 없으므로 제거한다.

③ Node $\{d, f\}$가 2 차로 바뀌었기 때문에 이와 연결된 branch, (g, d), (d, e), (e, f), (f, k)는 H에 포함되어야 한다.

이미 새로운 cycle, C가 구성되었음을 볼 수 있다. 여기에 부가적인 branch 를 첨가하면 차수가 증가하기 때문에 Hamiltonian cycle 을 구성할 수 없다.

Problem statement (*Traveling salesman problem: TSP*, 순회판매원 문제): Start at a city, visit every city (node) exactly once, and return to the initial city. We want to find the route with minimum distance.

TSP 는 가중치 graph, G에서 최소길이를 갖는 Hamiltonian cycle 를 찾는 것이다.

Example 8.4

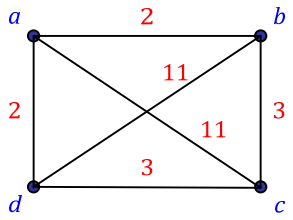


Fig. 8.12

(a) 위 graph, G에서 cycle $C:(a,b,c,d,a)$는 Hamiltonian cycle 이며 가중치의 총합은 10이다. 다른 경로를 선택하면 가중치가 11인 branch 를 포함해야 하기 때문에 전체 가중치의 합이 늘어나게 되어, cycle C가 TSP 의 해임을 알 수 있다.

Salesman-A 는 서울에 거주하고 있다. A 에게 해외에 있는 9개 고객을 방문 상담하는 임무가 부여되었다. 어느 회사든 그렇듯이 출장비가 빠듯하기 때문에, 가장 비용이 적게 드는 경로를 찾아야 한다. 모든 순서를 고려할 경우, $9! = 362{,}880$가지 경우의 수를 모두 고려해야 한다. 순회판매원 문제는 NP-복잡도를 갖는 가장 어려운 문제로 남아있다.

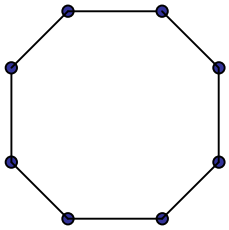


Fig. 8.13 Ring model for parallel computation

(b) Fig.8.13 은 병렬연산을 위한 ring model 을 보여주고 있다. Simple cycle 로 구성되어 있음을 볼 수 있다. Processor 를 의미하는 각 node 쌍을 하나의 branch 가 연결하고 있으며, 서로 data 를 직접적으로 주고 받을 수 있음을 표시한다.

Question: When we model an n-cube with a ring model? That is, when the n-cube contains a simple cycle with 2^n-nodes.

Claim that the n-cube $(n \geq 2)$ has a Hamiltonian cycle, if and only if there is a sequence,

(8.1) $\quad s_1, s_2, \ldots, s_{2^n}$

where s_i is an n-bit string, satisfying

(1) s_i and s_{i+1} differ in exactly one bit, $i = 1, 2, \ldots, 2^n$

(2) s_1 and s_{2^n} differ in exactly one bit.

A sequence in eq.8.1 is called a *Gray code*. $n \geq 2$일 때, Gray code 는 Hamiltonian cycle 과 일치한다. 편의상, $n = 1$일 때, Gray code$(0,1)$은 경로 $(0,1,0)$에 대응한다. 이 경로는 $(0,1)$-branch 가 두 번 반복되기 때문에 cycle 을 구성하지는 않는다.

Theorem 8.6 Let G_1 denote the sequence $(0,1)$. Define

(8.2) $\quad G_n = (G'_{n-1} : G''_{n-1})$

which is a concatenation of two sequences G'_{n-1} and G''_{n-1}, where

(a) G^R_{n-1}: sequence G_{n-1} arranged in reverse order

(b) G'_{n-1}: sequence G_{n-1}의 각 성분의 앞에 $'0'$을 첨가하여 구성한 sequence

(c) G''_{n-1}: sequence G^R_{n-1}의 각 성분의 앞에 $'1'$을 첨가하여 구성한 sequence

Then, G_n is a Gray code for all $n \geq 2$.

Corollary: The n-cube has a Hamiltonian cycle, $\forall\, n \geq 2$.

Table 8.1 Gray code

G_1	0	1						
G_1^R	1	0						
G'_1	00	01						
G''_1	11	10						
G_2	00	01	11	10				
G_2^R	10	11	01	00				
G'_2	000	001	011	010				
G''_2	110	111	101	100				
G_3	000	001	011	010	110	111	101	100
G_3^R	100	101	111	110	010	011	001	000
G'_3	0000	0001	0011	0010	0110	0111	0101	0100
G''_3	1100	1101	1111	1110	1010	1011	1001	1000
G_4								

8.4 A shortest path algorithm

연결된 가중치 graph, G에서 node-a와 node-z를 연결하는 가장 짧은 (최소 가중치 합) 경로를 찾는 문제를 고려해 보자. Dijkstra algorithm 은 최단 경로 문제를 푸는 가장 효율적인 것 중 하나로 알려져 있다. Branch-(i,j)의 가중치를 $w(i,j)$라 하고, node-v을 $L(v)$로 표기한다.

Dijkstra algorithm 은

(1) 먼저 각 node-v에 node-a로부터의 최단 경로 길이, $L(v)$를 할당한다. Algorithm 의 시작점에서는 최단경로를 알 수 없기 때문에, $L(a) = 0$, $L(v) = \infty, \forall v \neq a$로 설정한다. 무한대 값은 현 시점에서 최단경로길이를 모른다는 것을 의미한다. Algorithm 이 종료되었을 때, $L(v)$는 node-a에서 node-v까지의 최단경로길이를 나타나게 되고, 경로가 존재하지 않으면 여전히 무한대로 남는다.

(2) Node-a에서 node-v까지의 최단경로길이 $L(v)$를 알고 있고, node-v에서 node-u를 연결하는 $w(u,v)$인 branch 가 있으면, node-a에서 node-u까지의 최단 경로 길이는 $L(v) + w(v,u)$가 되며, 이 거리가 $L(u)$보다 작으면, $L(u)$ 값을 새로운 값으로 바꾼다.

Algorithm 8.1 (Dijkstra algorithm) Find the shortest path and its length from node-a to node-z.

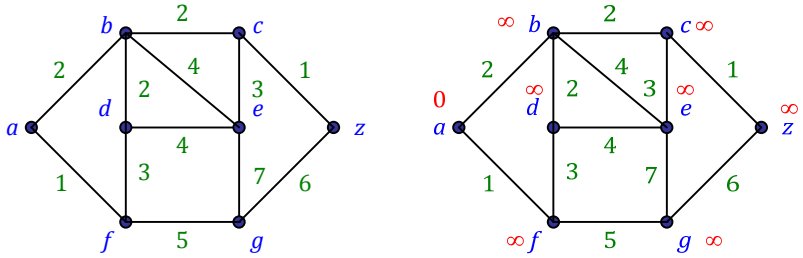
input: a connected weighted graph with weights

output: $L(z)$

$dijkstra(w, a, z, L)$ {

$L(a) = 0$
for all nodes $x \neq a$
    $L(x) = \infty$
$T = set\ of\ all\ nodes$ // T is the set of all nodes whose shortest distance from node-a
                           // has not been found
while $(z \in T)$ {
    choose $v \in T$ with minimum $L(v)$
    $T = T - \{v\}$
    for each $x \in T$ connected to node-v
        $L(x) = \min\{L(x), L(v) + w(v,x)\}$
}
}

Example 8.5



(a) (b) 초기화

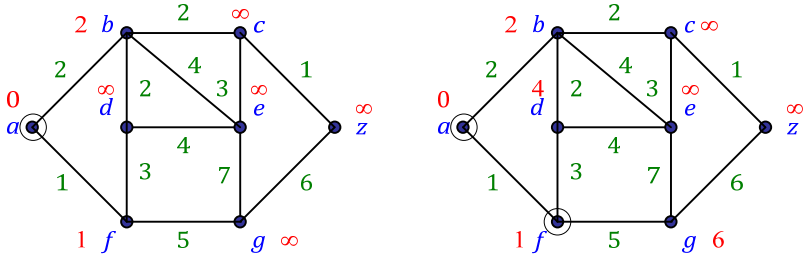
Fig. 8.14 (a) Connected weighted graph, (b) 초기화

(a) 초기화 단계에서 시작점인 node-a를 제외한 나머지 모든 node 에 무한대의 경로길이 $(L(v) = \infty, \forall v \neq a)$를 할당한다. 이때 $T = \{a, b, c, d, e, f, g, z\}$.

(b) 1st while-loop: T에 소속된 node 중에서 $L(a) = 0$으로 최소값을 가지므로, $v = a$로 선택하고 T를 $\{b, c, d, e, f, g, z\}$로 변경한다. Node-a를 고정한다. 그림에는 고정된 node 에 동그라미를 표시하였다. Node-a와 연결된 (un-circled) node 는 $\{b, f\}$이며, 이들의 최소경로길이를 계산한다.

$$L(b) = min\{L(b), L(a) + w(a,b)\} = min\{\infty, 0 + 2\} = 2,$$
$$L(f) = min\{L(f), L(a) + w(a,f)\} = min\{\infty, 0 + 1\} = 1$$

변경된 최소경로길이로 $L(b)$와 $L(f)$ 값을 변경한다.

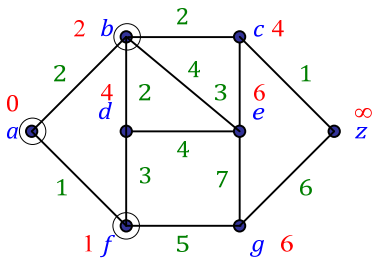


(c) 1st while-loop (d) 2nd while-loop

(c) 2nd while-loop: T에 소속된 node 중에서 $L(f) = 1$로 최소값을 가지므로, $v = f$로 선택하고 T를 $\{b, c, d, e, g, z\}$로 변경한다. Node-f를 고정하고 동그라미를 쳐 표시한다. Node-f와 연결된 (un-circled) node 는 $\{d, g\}$이며, 이들의 최소경로길이를 계산하고 변경된 값을 반영한다.

$$L(d) = min\{L(d), L(f) + w(f,d)\} = min\{\infty, 1 + 3\} = 4,$$

$$L(g) = min\{L(g), L(f) + w(f,g)\} = min\{\infty, 1+5\} = 6$$



(e) 3rd while-loop

(c) 3rd while-loop: T에 소속된 node 중에서 $L(b) = 2$로 최소값을 가지므로, $v = b$로 선택하고 T를 $\{c, d, e, g, z\}$로 변경한다. Node-b와 연결된 (un-circled) node 는 $\{d, c, e\}$이며, 이들 node 에서의 최소경로길이 값을 다음과 같이 변경한다.

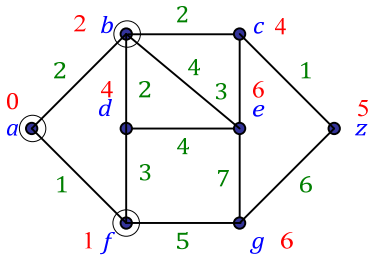
$$L(d) = min\{L(d), L(b) + w(b,d)\} = min\{4, 2+2\} = 4,$$

$$L(c) = min\{L(c), L(b) + w(b,c)\} = min\{\infty, 2+2\} = 4,$$

$$L(e) = min\{L(e), L(b) + w(b,e)\} = min\{\infty, 2+4\} = 6$$

Node-b를 동그라미 쳐서 고정한다.

(d) 이러한 단계를 반복하면, 최종적으로 길이 5인 최단경로 (a, b, c, z)를 얻을 수 있다.



(f) 종료 후