통신네트워크 Project 2

2018142059 김서영

◆ 실행 결과 (Data1, Source Node A의 경우)

- Dijkstra 알고리즘

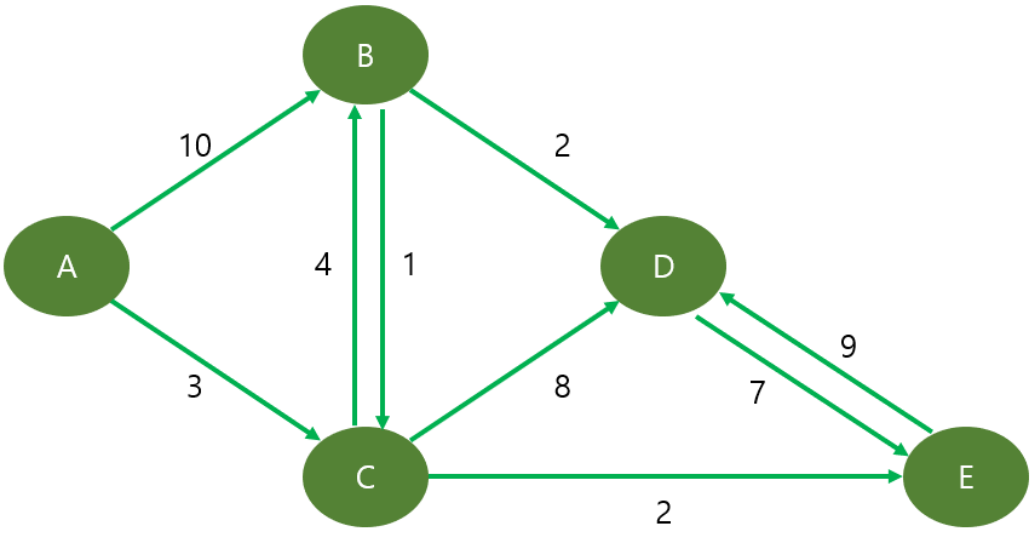
```
A file with graph data has been open.

   A graph has been created with 5 nodes.
   An edge has been created (A -> B, weight: 10)
   An edge has been created (A -> C, weight: 3)
   An edge has been created (B -> C, weight: 1)
   An edge has been created (B -> D, weight: 2)
   An edge has been created (C -> B, weight: 4)
   An edge has been created (C -> D, weight: 8)
   An edge has been created (C -> E, weight: 2)
   An edge has been created (D -> E, weight: 7)
   An edge has been created (E -> D, weight: 9)

   => 5 nodes & 9 edges have been created.

        input source (A, B or ...): A

        A to B,    cost:  7    path: B <- C <- A
        A to C,    cost:  3    path: C <- A
        A to D,    cost:  9    path: D <- B <- C <- A
        A to E,    cost:  5    path: E <- C <- A
ubuntu@myubuntu:~/my_share/pj2/Dijkstra$
```

- Bellman-ford 알고리즘

```
   An edge has been created (A -> B, weight: 10)
   An edge has been created (A -> C, weight: 3)
   An edge has been created (B -> C, weight: 1)
   An edge has been created (B -> D, weight: 2)
   An edge has been created (C -> B, weight: 4)
   An edge has been created (C -> D, weight: 8)
   An edge has been created (C -> E, weight: 2)
   An edge has been created (D -> E, weight: 7)
   An edge has been created (E -> D, weight: 9)

   => 5 nodes & 9 edges have been created.

        input source (A, B or ...): A

        A to B,    cost:  7    path: B <- C <- A
        A to C,    cost:  3    path: C <- A
        A to D,    cost:  9    path: D <- B <- C <- A
        A to E,    cost:  5    path: E <- C <- A

        A to B,    cost:  7    path: B <- C <- A
        A to C,    cost:  3    path: C <- A
        A to D,    cost:  9    path: D <- B <- C <- A
        A to E,    cost:  5    path: E <- C <- A
ubuntu@myubuntu:~/my_share/pj2/Bellman$
```

◆ **알고리즘 동작 설명**

- **Data 1**



- **Dijkstra 알고리즘**

Dijkstra 알고리즘은 centralized 알고리즘으로, 네트워크 정보(네트워크 내의 모든 edge의 weight)를 알고 이를 바탕으로 forwarding table을 작성한다.

먼저 전체 네트워크에 대한 Forwarding Table을 initialize 한다. 각각의 node에 대해 source로부터의 cost는 무한, 선행 노드는 없다고 지정하고, source node에 대해서 cost는 0, 선행 노드는 source node 자기 자신으로 지정한다.

Step	S	D(A)	D(B)	D(C)	D(D)	D(E)
initial	-	0, A	-1, ∞	-1, ∞	-1, ∞	-1, ∞

Source node부터 차례대로, Source node로부터의 거리가 가장 짧은 노드(cost가 가장 작은 노드)부터 해당 노드에서 다음으로 연결되는 모든 노드들에 대해 relaxation을 진행한다. 예를 들어 A와 A와 연결된 B에 대한 relaxation을 진행하면, A의 cost + B까지의 weight=10와 B의 현재 cost=∞를 비교하여 둘 중 더 작은 값을 B의 cost로 업데이트하고, B의 선행 노드를 A로 지정한다. 같은 방법으로 A 다음으로 연결된 또다른 노드인 C에 대해서도 마찬가지로 relaxation을 진행하면 다음과 같이 정리할 수 있다.

Step	S	D(A)	D(B)	D(C)	D(D)	D(E)
initial	-	0, A	-1, ∞	-1, ∞	-1, ∞	-1, ∞
0	A	0, A	10, A	3, A		

Forwarding Table에 없는 노드 B, C, D, E 중 source node A로부터 가장 가까운 노드는 cost가 3인

C이다. 따라서 C에서 연결되는 노드 B, D, E에 대해 relaxation을 진행한다. B의 기존 cost는 10이고, C의 cost + C에서 B로의 weight=3+4=7로 B의 기존 cost인 10보다 작으므로, B의 cost를 7로, 선행 노드를 C로 업데이트한다. 이를 D, E에도 같은 방법으로 적용하면 다음과 같다.

Step	S	D(A)	D(B)	D(C)	D(D)	D(E)
initial	-	0, A	-1, ∞	-1, ∞	-1, ∞	-1, ∞
0	A	0, A	10, A	3, A		
1	AC		7, C		11, C	5, C

Step 1을 거쳤을 때 Forwarding Table S에 없는 노드 B, D, E중 cost가 가장 작은건 E 노드이므로, 다음 step에서는 E에서 relaxation을 진행하고, E를 forwarding table에 넣는다.

Step	S	D(A)	D(B)	D(C)	D(D)	D(E)
initial	-	0, A	-1, ∞	-1, ∞	-1, ∞	-1, ∞
0	A	0, A	10, A	3, A		
1	AC		7, C		11, C	5, C
2	ACE				11, C	

E에 대해 relaxation을 진행할 때, E에서 다음으로 연결되는 노드는 D가 유일하다. 하지만, D의 기존 cost 11이 E의 cost+E에서 D로 갈 때의 weight=5+9=14보다 작기 때문에 D의 cost는 업데이트 되지 않고 relaxation이 끝난다. 다시 S에 존재하지 않는 노드 B, D 중에서 cost가 더 작은 B에 대해 relaxation을 진행하고, B를 Forwarding table에 넣게 된다.

Step	S	D(A)	D(B)	D(C)	D(D)	D(E)
initial	-	0, A	-1, ∞	-1, ∞	-1, ∞	-1, ∞
0	A	0, A	10, A	3, A		
1	AC		7, C		11, C	5, C
2	ACE				11, C	
3	ACEB				9, B	

D의 기존 cost가 B에서 D로 가는 weight+B의 cost보다 크기 때문에, D의 선행 노드와 cost가 update 된다.

Step	S	D(A)	D(B)	D(C)	D(D)	D(E)
initial	-	0, A	-1, ∞	-1, ∞	-1, ∞	-1, ∞
0	A	0, A	10, A	3, A		
1	AC		7, C		11, C	5, C
2	ACE				11, C	
3	ACEB				9, B	
4	ACEBD					

변화한 D의 cost로 인해 cost가 새로 바뀌는 노드가 없으므로, Forwarding Table S에 D를 포함한 모든 노드가 들어가게 되어 알고리즘이 종료된다.

- **Bellman-ford 알고리즘**

Bellman-ford 은 decentralized 알고리즘으로, 한 노드는 그 바로 주변 노드들의 cost들과 그 사이의 weight만을 아는 상태에서 cost를 계산한다. 따라서 앞의 Dijkstra 알고리즘과 달리 어떤 노드 x에서 목적지 노드 y까지의 경로까지의 cost를 다음과 같은 Bellman-Ford equation으로 모든 노드들의 cost를 각각 계산하는 것을 반복하는 iteration을 통해 forwarding table을 완성한다.

$$D_x(y) \leftarrow \min_v\{c(x,v) + D_v(y)\}$$

Code에서는 최적화 이전에 initialization을 진행한다. 이는 앞의 Dijkstra 알고리즘에서의 initialization과 같이, 모든 노드들의 cost를 무한으로 두고, 선행 노드를 없다고 정하고, source node의 cost를 0으로, 선행 노드를 source node 그 스스로로 지정한다.

Source node의 cost만 0이고, 그 외의 노드들의 cost는 무한이기 때문에, 첫번째 iteration에서는 source node에서만 relaxation의 cost update가 실행된다. Source node A의 다음으로 연결된 노드 B, C의 cost는 다음과 같이 update된다.

Iteration	A	B	C	D	E
Initial	0, A				
0	0, A	10, A	3, A		

그 다음 iteration에서는 cost가 무한이 아닌 세 노드 A, B, C에 대해 relaxation을 진행한다. A는 선행 노드가 A 스스로이므로 new cost가 변화하지 않고, B의 경우 C는 기존의 old cost가 B를 거치는 new cost보다 더 작으므로 변화하지 않고, D는 B의 old cost에 B->C weight를 더한 12로 new cost가 변화한다. C의 경우, C에서 갈 수 있는 노드 B, D, E에 대해 relaxation을 진행한다. C를 거쳐 B로 가는 경로의new cost가 A에서 B로 곧장 가는 경로의 old cost보다 작기 때문에 B의 new cost가 7로 변화하고, 선행 노드도 C로 업데이트된다. 같은 방식으로 D의 new cost도 11, E의 new cost도 5로 업데이트 된다.

Iteration	A	B	C	D	E
Initial	0, A				
0	0, A	10, A	3, A		
1 - A	0, A				
1 - B					
1 - C		7, C		11, C	5, C

이제 모든 노드들이 무한이 아닌 cost값을 갖기 때문에 각각의 노드들에 대해 모두 relaxation을

진행하여야 한다. B와 D가 연결된 edge에서 relaxation을 진행할 때, 기존 old cost 11보다 B의 old cost + B에서 D로 가는 edge의 weight = 9가 더 작기 때문에 new cost = 9로 업데이트된다. 그 외 다른 노드에서의 변화는 존재하지 않는다.

Iteration	A	B	C	D	E
Initial	0, A				
0	0, A	10, A	3, A		
1 - A	0, A				
1 - B					
1 - C		7, C		11, C	5, C
2 – A					
2 – B				9, B	
2 - C					
2 – D					
2 - E					

이후 한 번 더 iteration이 진행되지만, 그에 따른 cost 변화는 없다.

Weight update 과정

```
bool Bellman_Ford_update(pGraph pG, int s, int c)
{
	int l;
	int i, j, k;

	//weight from certain node is changed
	gChange_weight(pG,c);
	for (l = 0; l < gGet_deg(pG,s); l++)
	{
		//reset cost of nodes that are connected to node from source node
		while(gGet_prenode(pG,c)!=s&&gGet_prenode(pG,c)!=-1)
		{
			c = gGet_prenode(pG,c);
			gSet_newcost(pG,c,INF);
		}

	}
	gRenew_cost(pG);

	for (i = 0; i < gGet_N(pG) -1; i++)
	{
		for (j = 0; j < gGet_N(pG); j++)
			for (k = 0; k < gGet_deg(pG, j); k++)
				Relaxation(pG, j, k);
		gRenew_cost(pG);
	}

	for (j=0; j < gGet_N(pG); j++)
		for (k=0; k < gGet_deg(pG, j); k++)
			if (Exist_neg_Cycle(pG, j, k))
				return false;

	return true;
}
```

Bellman-Ford 알고리즘에서 Weight update 함수의 code는 캡쳐와 같다. 노드 C에서 다른 노드로 나가는 edge들의 weight들이 변화하면, 이에 따라 cost를 update하기 위해 source node에서 노드 C까지 가는 경로에 있는 모든 node들의 cost를 무한으로 수정한다.

이는 edge의 weight가 증가하여 cost를 업데이트할 때 이러한 조치 없이 진행한다면 해당 edge를 거치는 경로의 cost를 계속 이용하여 너무 많은 iteration을 반복하게 된다. 따라서 이후에 weight가 변화한 edge를 지날 수 있는, 노드 C를 거치는 경로들의 cost를 초기화하여 다시 cost를 산정하게 하는 것이다.

Source node에서 C까지의 경로 상의 노드들의 cost를 초기화한 이후, 모든 노드들의 edge들에 대하여 relaxation을 진행하는 것을 (node 수-1)만큼 반복하여 전체 cost의 update를 진행한다.