알고리즘 HW #3

컴퓨터공학부
2013-11431 정현진

1. 수업 시간에 배운 topological sorting algorithm(DFS를 변형한 버전)이 제대로 결과를 낸다는 것을 증명하라.

topological sorting의 경우 vertex i, j가 있고 i->j로 향하는 간선이 있을 때, i는 항상 j보다 앞서서 방문하도록 해야 한다. 이 때 수업 때 배운 알고리즘을 기준으로 한다면, 방문된 순서가 늦은 순서대로 출력이 먼저 된다. 즉, i->j일 때 j가 먼저 출력되고 i가 나중에 출력되어야 한다.

모순을 위해 어떤 vertex i, j가 있고 i->j로 향하는 edge가 있는 상황에서 i가 j보다 먼저 출력되는 상황이 있다고 가정하자. 우선 DFS에서 j를 먼저 방문한 경우를 생각해보면, topological sorting은 DAG를 기준으로 하므로 j --*> i 의 경로가 존재하지 않기 때문에 i를 방문할 수 없어 모순이 된다. i를 먼저 방문한 경우에는 i->j의 edge가 존재하기 때문에 i에서 j로 DFS를 호출해 방문을 하고, j의 방문 순서가 i보다 늦기 때문에 항상 j가 i에 앞서 출력이 되게 된다. 두 경우를 모두 고려해 볼 때 어떠한 vertex i, j가 있을 때 i->j의 간선이 존재하는 경우 i가 j보다 먼저 출력되는 경우는 없다고 할 수 있다.

이러한 상황을 전체로 확장시켜보면, 모든 경우에서 방문 순서가 늦을수록 출력이 먼저 된다는 것을 알 수 있고 전체 출력의 순서는 topological sorting의 역순이 되어 제대로 작동한다.

2. Tree를 이용한 집합의 표현에서 Path-Compression을 이용한 Find-Set 알고리즘은 아래와 같다. 이를 recursion을 사용하지 않는 버전으로 바꾸어 보아라.

```
Find-Set(x)
{
  If (p[x] ≠ x) then p[x] ← Find-Set(p[x]);

  return p[x];
}
```

이를 recursion을 사용하지 않는 버전으로 바꾸어 보면 다음과 같이 표현할 수 있다.

```
Find-Set(x) - Without Recursion
{
    i = 0;
    result = x;

    while(p[result] ≠ result):
        List.add(result);           // Find-Set 수행 도중 거쳐간 node들을 담는다.
        result ← p[result];

    while(!List.isempty()):         // path-compression
        tmp = List.remove();
        p[tmp] = result;

    return result;
}
```

3. 그래프 G=(V, E)의 edge들의 weight이 (1, 100)의 범위에서 uniform distribution을 이룬다. 당신이라면 minimum spanning tree를 찾기 위해 어떤 알고리즘을 사용하겠는가? 당신의 답에 대한 이유도 같이 설명하라.

Minimum Spanning Tree를 찾기 위한 알고리즘은 프림 알고리즘과 크루스칼 알고리즘이 있다. 두 알고리즘은 모두 O(|E|log|V|)의 수행 시간을 가진다. 이 중에서 크루스칼 알고리즘을 사용하려고 한다. 크루스칼 알고리즘의 작동 과정을 간단하게 기술해보면 다음과 같이 나타낼 수 있다.

1. T = empty 로 초기화.
2. 각각 1개씩의 vertex를 담고 있는 n개의 singleton set을 초기화
3. edge들을 non-decreasing weight 순서로 정렬하고 Q에 삽입.
4. T가 |v| - 1개 보다 적은 수의 edge를 가지고 있을 동안, Q에서 최소 cost edge {u,v}를 선택하고 Q에서 삭제한다. 만약 u와 v가 다른 set에 속해 있다면 T = T U {{u, v}}가 되어 u와 v를 담고 있는 두 set을 합친다.

이 과정들의 수행시간을 살펴보면, 1번은 상수, 2번은 O(n), 3번은 O(|E|log|E|) = O(|E|log|V|), 4번은 tree 구조와 여러 가지 휴리스틱을 사용하면 O($|E|\log^*|V|$)의 수행 시간을 가지게 되어 결국 3번의 edge 정렬 시간이 가장 수행시간을 많이 차지하는 부분인 것을 알 수 있다.
이 때 edge들의 weight가 (1, 100)의 범위에 있는 경우에는 3번의 edge 정렬을 radix sort, bucket sort 등 linear time sorting 방식을 이용할 수 있게 된다. 그러면 최악의 경우 정렬 시간을 O(|E|)로 줄일 수 있으므로, linear time sorting을 사용할 때 크루스칼 알고리즘의 전체 수행 시간을 O($|E|\log^*|V|) \approx O(|E|)$까지 줄일 수 있게 된다.
반면에 프림 알고리즘의 경우 priority queue를 사용하기 때문에 이 문제 상황에서 기존의 알고리즘보다 수행 시간을 효율적으로 상승시킬 수 없다.
따라서 이 문제의 상황에서 크루스칼 알고리즘을 사용할 것이다.

4. Weighted directed graph G=(V, E)가 적어도 하나의 negative weight cycle을 갖고 있다.
이 그래프에서 임의의 negative weight cycle을 하나 찾아 그 cycle에 속하는 vertex들을 나열하는 알고리즘을 기술하고, 그 복잡도를 밝히라. 또 이 알고리즘이 제대로 작동한다는 것을 증명하라.

알고리즘을 기술하면 다음과 같다.
우선 알고리즘은 벨만-포드 알고리즘에 기반하고 있다. |V| - 1번의 loop를 돌며 relaxation을 한 후, V번째에 d[v] > d[u] + w(u,v)인 vertex v가 있는지 살펴본다. 그러한 경우가 있으면 v를 시작으로 prev[v]를 타고 올라가며 cycle을 구한다. cycle의 시작점(이자 끝점)을 저장하고 cycle을 거슬러 올라가며 찾은 뒤 출력한다.
이를 간단하게 pseudo-code로 나타내면 다음과 같다.

```
1.  Finding Negative-Weight-Cycle Algorithm:
2.    d[s] = 0;
3.    for each v in V - {s}:
4.      d[v] = infinite;
5.      prev[v] = NULL;
6.    // 여기까지 initialize.
7.
8.    // relaxation
9.    for i <- 1 to |v| - 1:
10.     for each (u,v) in E:
11.       d[v] = min{d[v], d[u] + w(u,v)};
12.       prev[v] = u;
13.
14.   // negative-cycle 찾기
15.   for each (u,v) in E:
16.     if(d[v] > d[u] + w(u,v)):
17.     {
18.       print("Negative Weight Cycle!");
19.       for each vertex in V:
20.         visited[vertex] = false;
21.       // negative cycle의 시작점(이자 끝점)을 찾는다.
22.       while(visited[v] == false):
23.         visited[v] = true;
24.         v = prev[v];
25.
26.       result.add(v);
27.       tmp = prev[v];
28.       // cycle의 시작점으로부터 previous vertex들을 거슬러 올라가며 cycle 저장.
29.       while(tmp != v):
30.         result.add(tmp);
31.         tmp = prev[v];
32.
33.       // cycle임을 명확하게 하기 위해 마지막에 시작점==끝점을 한번 더 담아준다.
34.       result.add(v);
35.     }
36.     else:
37.       result = empty;
38.
39.   return result;
```

이 알고리즘의 수행 시간을 살펴보자. 3~5줄의 초기화는 O(V)의 수행 시간이 소요된다. 9~12줄의 경우 V-1개의 vertex에서 인접한 Edge들을 모두 살펴보므로 O(|V||E|)의 수행 시간이 걸린다. 15~36줄은 만약 negative cycle이 없을 경우 O(E), 사이클이 존재하는 경우에는 최악의 경우 E번의 바깥 for문을 수행하고 그 중 if문에 탐색이 될 경우 v를 거슬러 올라가며 최악의 경우 V번의 vertex를 추적할것이므로 O(|V|+|E|)의 시간이 소요된다. 즉 전체 알고리즘의

수행 시간은 O(|V||E|)이다.

이제 이 알고리즘의 작동 증명을 해보자. 우선 벨만-포드 알고리즘에서 V-1번의 loop 수행을 하며 relaxation 한 후, V번째 수행 때 d[v] > d[u] + w(u,v)인 vertex가 존재하면 negative weight cycle이 탐색된다는 것을 증명하자.

negative weight cycle $(v_0, v_1, v_2, \cdots, v_{k(=v_0)})$ 이 있다고 하자. 그러면 $\sum_{i=1}^{k} w(v_{(i-1)}, v_i)$ < 0 이다.

만약 "Negative Weight Cycle!"이라는 출력이 나오지 않았다고 가정해보자. 그러면 if문의 조건에 맞지 않았음을 의미하고, i = 1, 2, 3, ..., k일 때 d[v_i] ≦ d[v_{i-1}] + w(v_{i-1}, v_i)이다.

이 때 $\sum_{i=1}^{k} d[v_i]$ ≦ $\sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$ 이 되는데, v_0~v_k가 cycle을 이루므로

$\sum_{i=1}^{k} d[v_{i-1}]$ = $\sum_{i=1}^{k} d[v_i]$ 이다. 따라서 $\sum_{i=1}^{k} d[v_i]$ ≦ $\sum_{i=1}^{k} d[v_i] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$ 가 되는데,

이 경우를 만족하려면 $\sum_{i=1}^{k} w(v_{(i-1)}, v_i)$ ≧ 0 이어야 한다. 하지만 정의에 의해

$\sum_{i=1}^{k} w(v_{(i-1)}, v_i)$ < 0이므로, 이는 모순임을 알 수 있다. 따라서 임의의 negative weight cycle이 존재한다면 항상 if문의 조건에 걸리는 것을 알 수 있다.

이제 v를 기준으로 previous vertex를 거슬러 올라갈 때 항상 negative weight cycle의 경로를 찾을 수 있는가?에 대한 증명을 해보자. 총 3가지의 경우가 있다.

1) negative weight cycle을 발견한다.
2) positive weight cycle을 발견한다.
3) cycle을 발견하지 못한다.

우선 1)의 경우는 우리가 원하는 경우이므로 넘어간다.
2)의 경우는 벨만-포드 알고리즘의 relaxation 과정에서 positive weight cycle이 존재할 수 없으므로 발생할 수 없는 상황이다.
3)의 경우를 생각해보자. cycle을 발견하지 못하기 때문에 v부터 previous vertex들을 거슬러 올라가다 보면 언젠가는 출발점인 s에 도달할 것이다. 그 뜻은 s --*> v로 가는 경로가 사이클을 만들지 않는 acyclic path라는 것이다. 이 경우 v에서 relaxation이 일어났다는 뜻은 기존에 업데이트된 최단 경로 d[v]보다 더 짧은 경로가 존재한다는 뜻이다. 하지만 이러한 경로가 존재했다면 앞서 |V| - 1번의 relaxation 과정에서 찾아내서 업데이트를 했을 것이다. 따라서 3)의 경우도 모순인 것을 알 수 있다.

따라서 항상 v를 시작으로 previous vertex들을 거슬러 올라가다 보면 negative weight cycle을 찾을 수 있음을 알 수 있다.

5. Dijkstra algorithm을 가장 짧은 경로는 물론 두 번째로 짧은 경로의 길이까지 같이 구할 수 있도록 변형해 보아라. 당신의 알고리즘이 제대로 작동한다는 것을 설명하라.(informal한 설명도 okay) 수행시간도 밝히라.

시작점 s로부터 출발해 임의의 vertex v까지 도달하는 2번째 최단경로(앞으로 SSP라 부르겠다.)를 p(v)라고 하자. 이 때 p(v)는 다음과 같이 나타낼 수 있다.

p(v): p_1(u) + u->v

p(v)가 s부터 v까지의 SSP이므로 p1(u)는 s부터 u까지의 SP(Shortest Path) 혹은 SSP가 된다. 만약 p1(u)가 SP도 아니고 SSP도 아니라고 하자. 이 경우 s부터 u까지의 SP가 되는 p2(u)나 SSP가 되는 p3(u)가 존재할 것이다. 이 경우 p2(u) + u->v나 p3(u) + u->v는 모두 p1(u) + u->v의 길이보다 짧은 길이를 가지게 되므로 p(v)는 SSP가 아니게 된다. 따라서 p(v)가 SSP라고 했던 가정에 모순이므로 p1(u)는 SP 혹은 SSP임을 알 수 있다.
이 때 p(v)의 길이는 p1(u)의 길이에 w(u,v)를 더한 것이 된다. p1(u)는 SP일 수도, SSP일 수도 있으므로 다음과 같이 나타낼 수 있다.

d1(v)를 s부터 v까지 SP의 길이, d2(v)를 s부터 v까지 SSP의 길이라고 하자. 그러면 d2(v)는 d1(u)가 확정된 u에 대해 d1(u) + w(u,v) 이거나 d2(u)가 확정된 u에 대해 d2(u) + w(u,v)가 된다.

위의 설명을 기반으로 다익스트라 알고리즘을 변형해 보았다. 알고리즘의 뼈대는 기존의 다익스트라 알고리즘과 비슷하지만 각각 vertex들의 최단 경로를 담고 있는 d1과 2번째 최단 경로를 담고 있는 d2를 가지고 있다는 점이 다르다. 그리고 각각 vertex들은 d1[v]의 값을 기준으로 Q에 들어간 뒤 d2[v]의 값을 기준으로 다시 Q에 들어가게 함으로써 위의 증명의 내용을 구현하였다.

```
Second-Shortest Dijkstra Algorithm:
// 만약 존재하는 경로가 하나 뿐이라면 second shortest의 길이는 inf
// 같은 길이의 최소 경로가 여러개일 경우 second shortest라고 하지 않는다.
  d1[s] = 0;
  d2[s] = 0;
  create Q;

  for(each vertex u in V):
    if u ≠ source:
      d1[u] = inf;
      d2[u] = inf;
      u.queue_value = d1[u];
      add u to Q;

  // 여기까지 d1, d2, Q 초기화.

  while(Q != empty):
    u = extractMin(Q);  // 최솟값 추출
    for each neighbor v of u in Q:
      // d1 값이 queue_value일 경우는 d1값으로 업데이트하고
      // d2값이 queue_value일 경우는 d2값으로 업데이트하기 위해 u.queue_value와 비교.
      if(d1[v] > u.queue_value + w(u, v)):
        d2[v] = d1[v];
        d1[v] = u.queue_value + w(u,v);
        update priority queue Q with regard to v;
      else if(d2[v] > u.queue_value + w(u, v)):
        d2[v] = u.queue_value + w(u,v);
        update priority queue Q with regard to v;
      // d1값이 Q의 기준값이었으면 d2값을 기준으로 다시 Q에 넣음.
      if(u.queue_value == d1[u]):
        u.queue_value = d2[u];
        add u to Q;

  return d1,d2;
```

이 알고리즘의 수행 시간을 구하는 것은 간단한데, priority queue를 업데이트 하는 부분이 최악의 경우 2*E번 수행되고, vertex들이 Q에 2번씩 들어가므로 extractMin이 2*V번씩 수행된다는 점이 기존과 다르다. 즉 이 알고리즘의 수행 시간은 O(2*(|V|+|E|)log|V|) = O((|V|+|E|)log||V|)이다.

6. Weighted directed graph의 모든 edge weight이 {1, 2, ..., W}에 속할 때 single-source shortest path problem을 위한 Dijstra's algorithm은 O((|V|+|E|)logW) time에 해결될 수 있음을 보여라.

다익스트라 알고리즘의 경우 점근적 수행 시간에 크게 영향을 미치는 부분은 while문을 반복적으로 돌면서 우선 순위 heap에서 최솟값을 뽑는 extractMin과, d[v](시작점으로부터 vertex ‘v’까지의 최소 거리)의 값이 변하게 될 경우 힙을 업데이트 해 주는 부분이다. 이 때 heap은 최대 vertex 수만큼의 원소를 가지므로 extractMin은 모든 vertex에 대해 최대 log|V|만큼의 수행, 즉 O(|V|log|V|)만큼의 수행을 하고, 힙을 업데이트 하는 부분(updateHeap이라고 하자)은 모든 edge에 대해 최대 log|V|만큼의 수행을 해 O(|E|log|V|)의 수행을 하게 된다. 따라서 전체 수행 시간은 O((|V|+|E|)log|V|)가 된다.

이제 알고리즘의 수행 시간을 O((|V|+|E|)logW)로 만들기 위해 우선순위 힙의 구조를 바꾸어 보자. 임의의 vertex들이 있을 때 시작점으로부터의 최단 경로 거리가 같은 vertex들은 힙에서 같은 원소에 속한다고 하자(예를 들어 linked list와 같은 자료 구조를 통해 d값이 같은 vertex들을 같이 묶는다고 할 때 linked list 자체를 힙의 원소라고 하는 것이다.). 이렇게 구조를 바꾸었을 경우 힙의 최대 원소 개수가 O(W)를 만족시키는 것과, extractMin과 updateHeap의 수행 시간이 O(logW)를 만족시키는 것을 보인다면 수정한 알고리즘의 수행 시간이 O((|V|+|E|)logW)인 것을 증명하는 것이 될 것이다.

우선 힙의 원소 개수를 생각해보자. 처음에 초기화를 할 때는 시작점의 d값이 0이고 다른 모든 점은 inf이므로 원소의 개수는 2개가 될 것이다. 그 뒤 시작점을 방문해 인접한 vertex들의 d값을 업데이트 하는 경우, d값이 (1,…,W & inf)인 vertex들이 존재하므로 최대 W+1개의 원소를 가지게 된다. 그 뒤 최솟값인 1을 가지는 vertex들을 선택해 또 다시 d값을 업데이트 한다면 d값이 (2,…,W+1 & inf)인 vertex들이 존재할 것이고 d의 값이 1인 vertex가 모두 뽑히지 않았을 경우까지 고려한다면 최대 W+2개의 원소가 힙에 존재하게 된다. 이런 식으로 d값이 최소인 vertex들을 뽑아 방문을 진행할 때마다 최대 W+2개의 원소가 힙에 존재하게 된다. 그 뒤 모든 vertex의 d값을 한 번 이상 업데이트하게 된다면 d값이 inf인 vertex들이 없어져 최대 W+1개의 원소가 힙에 존재할 수 있을 것이다. 따라서 모든 경우 원소의 수가 최대 O(W)개를 만족한다는 것을 알 수 있다.

이제 extractMin의 수행 시간을 생각해보자. 만약 같은 d값을 가지는 vertex가 여러 개인 경우 그 중 하나를 뽑고 크기만 하나 줄이면 되므로 수행 시간이 상수일 것이다. 같은 d값을 가지는 vertex가 1개인 경우는 heap에서 그 원소를 삭제해주어야 하므로 최대 O(logW)의 시간이 걸릴 것이다. 즉 extractMin의 수행 시간은 O(logW)임을 알 수 있다.

그리고 updateHeap의 수행 시간을 생각해보자. 수정된 알고리즘의 경우 updateHeap의 작동은 힙에서 원하는 d값을 가진 원소가 어디 있는 지 찾은 뒤, 그 중에서 원하는 vertex를 찾고 업데이트하게 된다.

이 때 기존 다익스트라 알고리즘에서 사용하는, 힙에서 원소가 어디 있는지를 나타내는 hash table에 더해 한 원소에서 원하는 vertex가 어디 있는지를 나타내는 hash table을 사용하자. hash table의 insert, delete, find가 모두 O(1)에 작동하므로, 위의 동작 과정해서 update의 과정은 hash table들을 이용해 모두 상수 시간에 가능할 것이다. 단지 update 과정 중에서 원소의 vertex가 1개만 존재해 원소가 삭제되거나, 새로 update된 d값을 가진 원소가 존재하지 않아 새로 만들어야 할 경우는 heap에 새로운 원소를 delete와 insert를 하는 과정이 필요하게 된다. 이 경우의 수행 시간은 O(logW)가 된다. 즉 updateHeap은 O(logW)의 수행 시간을 가진다.

따라서 최대 V번의 extractMin 수행과 E번의 updateHeap 수행을 할 때 수정된 다익스트라 알고리즘의 수행 시간은 O((|V|+|E|)logW)임을 알 수 있다.