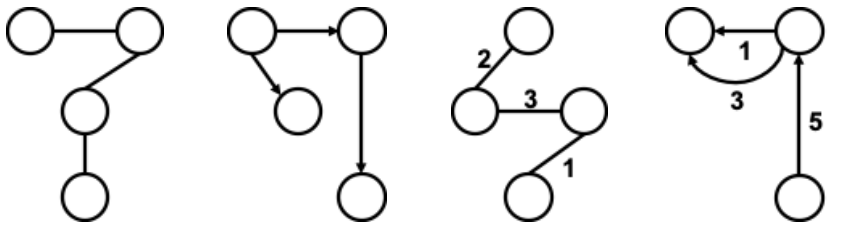
알고리즘및실습

제7장 그래프 탐색

1. 그래프

그래프는 트리보다 상위 개념으로 **노드**(node, vertex)와 노드를 연결하는 **간선**(edge)에 의해 정의된다. V가 노드의 집합이고 E가 간선의 집합일 때, 그래프 G는 $G = (V, E)$로 정의된다. 보통 주어진 그래프 G의 노드 수와 간선 수는 n과 m을 이용하여 나타낸다. 현재 우리는 GPS와 지도 앱을 활용하여 출발지에서 목적지까지 최단 경로를 확보할 수 있으며, 이를 이용하여 원하는 목적지까지 편하게 찾아 갈 수 있다. 보통 이와 같은 최단 경로는 그래프로 표현된 자료구조에서 한 노드에서 다른 노드까지의 최단 경로를 찾는 알고리즘을 이용한다.

1.1 그래프의 종류



(1) 무방향 그래프 (2) 방향 그래프 (3) 가중치 그래프 (4) 다중 그래프

<그림 7.1> 그래프의 종류

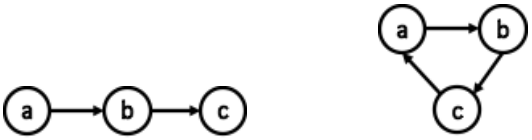
그래프는 간선의 방향성, 간선의 가중치 여부, 병행 간선(parallel edge) 존재 여부에 따라 그림 7.1와 같이 분류할 수 있다. 간선의 방향이 없으면 **무방향**(undirected) 그래프라 하고, 방향이 있으면 **방향** 그래프(directed graph, digraph)라 한다. 방향 그래프에서 간선을 다른 말로 아크(arc)라 한다. 방향 그래프에서 간선은 시작 노드와 끝 노드를 가지게 된다. 간선에 특정 값이 부여되는 경우도 있고 그렇지 않는 경우도 있다. 전자를 **가중치**(weighted) 그래프라 하고, 후자를 비가중치 그래프라 한다. 두 노드를 연결하는 간선이 여러 개 존재할 수 있는 그래프를 **다중 그래프**(multigraph)라 하고, 이와 같은 간선을 병행 간선이라 한다.

1.2 용어

인접(adjacent) 노드란 간선에 의해 연결된 노드들 말하며, 다른 말로 이웃 노드라 한다. 무방향 그래프에서는 간선에 의해 연결된 두 노드는 항상 서로의 인접 노드가 되지만 방향 그래프에서는 간선의 끝 노드는 시작 노드의 인접 노드이지만 그 반대는 끝 노드에서 시작 노드를 잇는 또 다른 간선이 없으면 인접 노드가 아니다. 인접성은 보통 두 노드 간의 관계를 나타내기 위해 사용되지만 간선과 노드 간의 관계를 나타내기 위해 사용할 수 있다. 방향 그래프에서 간선은 소스 노드의 인접 간선이 되지만 무방향에서 간선은 시작 노드와 끝 노드 모두의 인접 간선이

된다. 노드의 **차수**(order)란 노드와 연결된 간선의 수를 말하며, 방향 그래프에서는 진입 차수(indegree)와 진출 차수(outdegree)로 나누어 고려한다.

경로(path)란 두 개의 노드를 연결하는 일련의 노드를 말하며, 노드 a에서 b까지의 경로가 a, v_1, v_2, ..., v_k, b이면 그래프에 간선 (a, v_1), (v_1, v_2), ..., (v_k, b)가 존재한다는 것을 말한다. 비가중치 그래프에서 a에서 b까지의 경로의 길이는 간선의 수로 계산하며, 가중치 그래프에서는 경로를 구성하는 가중치의 합으로 계산한다. 주어진 경로에서 첫 번째 노드와 마지막 노드가 같으면 이 경로를 **주기**(cycle)라 한다. 단순 경로(simple path)는 한 노드를 두 번 거치지 않는 경로를 말한다. 따라서 단순 경로는 주기가 없는 경로이다.



(1) 약한 연결 방향 그래프 (2) 강한 연결 방향 그래프

<그림 7.2> 방향 그래프에서 연결 그래프

연결 그래프(connected graph)란 서로 다른 임의의 두 노드 사이에 경로가 항상 존재하는 그래프를 말한다. 임의의 노드에서 출발하여 깊이 우선이나 너비 우선 탐색을 통해 방문할 수 있는 노드의 집합이 항상 전체 노드 집합과 같으면 연결 그래프이다. 무방향 그래프의 경우에는 모든 노드를 검사하지 않고 하나의 노드에서 탐색을 해보아도 그래프의 연결 여부를 알 수 있다. n개 노드로 구성된 무방향 연결 그래프의 최소 간선의 수는 $n-1$이다. 방향 그래프에서는 약한 연결 그래프와 강한 연결 그래프 개념이 있다. 약한 연결은 연결 그래프가 아니지만 간선의 방향을 모두 제거하여 무방향 그래프로 변환하였을 때 연결 그래프가 되는 그래프를 말한다, 그림 7.2의 (1)은 약한 연결 그래프이지만 강한 연결 그래프는 아니다. 방향 그래프에서 루트 연결 그래프(rooted digraph) 개념도 있다. 루트 연결 그래프는 한 노드에서는 다른 모든 노드로 경로가 존재하는 그래프를 말하며, 이 노드를 해당 그래프의 루트라 한다.

완전 그래프(complete graph)란 모든 노드가 다른 노드의 인접 노드가 되는 그래프를 말한다. 따라서 간선의 시작과 끝 노드가 같은 간선을 고려하지 않고 다중 그래프가 아니면 완전 방향 그래프는 총 $n(n-1)$개의 간선이 있고, 완전 무방향 그래프는 $n(n-1)/2$개의 간선이 있다. 이것이 같은 조건의 그래프가 가질 수 있는 최대 간선 수이다. 참고로 다중 그래프를 고려하지 않고, 간선의 시작과 끝 노드가 같은 간선이 존재할 수 있다면 방향 그래프에 존재할 수 있는 간선의 최대 수는 n^2이고, 간선의 최소 수는 0이다. 그래프의 간선 수가 최대 간선 수에 가까우면 **밀집**(dense) 그래프라 하고, 반대로 최소 간선 수에 가까우면 **희소**(sparse) 그래프라 한다.

그래프 $G = (V, E)$가 있을 때 $V' \subseteq V$, $E' \subseteq E$인 그래프 $G'(V', E")$를 G의 부분 그래프라 한다. 그래프의 부분 그래프 중 원래 그래프의 모든 노드를 포함하는 트리를 **신장 트리**(spanning tree)라 한다. 신장 트리는 항상 $n-1$개의 간선으로 구성된다. 가중치 그래프에는 최소 신장 트리(MST, Minimum Spanning Tree)가 정의할 수 있으며, 최소 비용 신장 트리는 주어진 그래프의 신장 트리 중 간선의 가중치 합이 가장 작은 신장 트리를 말한다.

2. 그래프의 표현

그래프는 크게 **인접 행렬**(adjacent matrix)을 이용하거나 **인접 리스트**를 이용하여 표현한다. 물론 이 두 가지 방법으로 무조건 표현해야 하는 것은 아니다. 간선 목록만 유지할 수 있고, 2차원 배열로 지도 정보를 나타내는 데이터는 원 데이터를 그대로 활용할 수 있다.

인접 행렬의 공간 복잡도는 항상 $O(n^2)$이다. 무방향 그래프의 경우에는 항상 대칭 행렬이 되기 때문에 정보가 중복 표현되며, 이 때문에 희소 그래프이면 낭비되는 공간이 많다. 가중치 그래프를 인접 행렬로 표현할 경우에는 가중치 값의 범위에 따라 간선이 없는 경우를 어떤 값으로 표현할 것인지 결정해야 한다. 보통 음의 가중치가 없는

그래프에서는 -1을 사용하는 경우도 있고, 가중치의 범위가 제한적이면 매우 큰 수를 이용하는 경우도 있다. 또 그래프의 적용해야 하는 알고리즘에 따라 더 적합한 값이 있을 수 있다.

마땅한 값으로 간선이 없다는 것을 나타내기 힘든 경우에는 두 개의 배열이나 가중치와 존재 유무를 나타내는 두 개의 값으로 구성된 구조체 배열을 사용할 수 있다. 이 경우에는 공간 복잡도가 더 증가하므로 인접 행렬 대신에 인접 리스트를 이용하는 것이 더 효과적일 수 있다.

희소 그래프는 인접 리스트를 이용하여 표현하는 것이 공간 복잡도 측면에서 효과적이다. 인접 리스트의 공간 복잡도는 $O(n+m)$이다. 특히, 무방향 그래프는 역인접 리스트가 없어도 노드의 진출 정보와 진입 정보를 모두 알 수 있기 때문에 효과적이다. 반면에 방향 그래프에서 진입 노드 정보가 필요하면 역인접 리스트의 유지가 필요하다. 이 경우에는 역인접 리스트를 추가하는 것보다 인접 행렬을 사용하는 것이 더 효과적일 수 있다. 반면에 방향 그래프이지만 응용에 따라 역인접 리스트가 필요 없으면 여전히 인접 리스트를 사용하는 것이 효과적일 수 있다. 인접 리스트의 경우 리스트의 리스트로 나타낼 수 있고, 노드 레이블과 리스트 쌍으로 구성된 맵으로 표현할 수 있다.

3. 일반 그래프 탐색

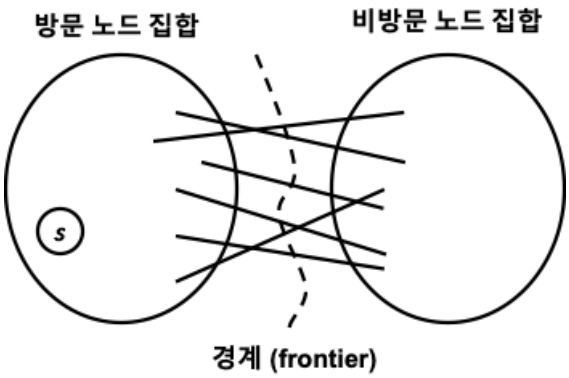
출발 노드부터 도달 가능한 모든 노드를 찾는 다음 문제를 생각하여 보자. 도달 가능하다는 것은 출발 노드부터 경로가 있는 노드를 말한다.

그래프 탐색 문제

- 입력. 그래프 G와 출발 노드 s
- 출력. s에서 출발하여 도달 가능한 노드 집합

알고리즘 7.1 일반 그래프 탐색 알고리즘

```
1: function SEARCH(G, s)
2:     visited := [false] × |G|
3:     visited[s] := true
4:     while true do
5:         select e = (u, v) where visited[u] = true and visited[v] = false
6:         if such e exists then visited[v] := true
7:         else break
8:     ret := [w for w in visited if visited[w]]
9:     return ret
```



<그림 7.3> 일반 그래프 탐색에서 간선 선택

그래프 탐색 문제를 해결하는 일반 알고리즘은 알고리즘 7.1과 같이 정의할 수 있다. 이 알고리즘은 그래프의 방향성과 무관하게 올바르게 동작한다. 이 알고리즘의 정확성은 다음과 같이 증명할 수 있다.

정리 7.1. 알고리즘 7.1에서 출발 노드가 s일 때 알고리즘이 종료되어 v가 답에 포함되기 위한 필요충분조건은 s에서 v까지 경로가 있어야 한다.

증명 필요충분조건이므로 양방향으로 증명을 해야 한다.

- $\Longrightarrow$: v가 답에 포함되면 s에서 v까지 경로가 있다.
  이 알고리즘은 경로를 따라 이동하므로 이 조건은 당연히 성립한다.
- $\Longleftarrow$: s에서 v까지 경로가 있으면 v는 답에 포함된다.
  이 조건은 모순을 이용하여 증명할 수 있다. s에서 v까지 경로가 있지만 v가 답에 포함되지 않았다고 하자. 노드 중에 최종적으로 답에 포함된 노드 집합이 E이고 $\bar{E}$가 답에 포함되지 않은 노드 집합이라 하자. 그러면 $s \in E$이고 $v \in \bar{E}$이다. 그런데 s에서 v까지 경로가 존재하기 때문에 E에서 $\bar{E}$로 가는 간선이 존재한다. 이 간선이 존재하면 알고리즘의 반복문은 종료할 수 없으므로 s에서 v까지 경로가 있지만 v가 답에 포함되지 않았다는 가정은 모순이 된다. 따라서 이 조건도 성립한다.

□

이 알고리즘을 구체화하기 위해서는 5번째 줄에서 조건을 만족하는 여러 간선 중 하나를 선택하는 기준이나 방법이 필요하다. 선택 기준 측면에서 생각하면 출발 노드가 있으므로 첫 번째로 선택 가능한 간선은 출발 노드에서 진출하는 간선 중 하나이다. 이때 하나를 선택하였다면 다음에 선택할 수 있는 것은 출발 노드와 첫 번째 선택한 간선의 끝 노드에서 출발하는 간선 중 하나이다. 그림 7.3와 같이 노드를 방문한 노드 집합과 아직 방문하지 않은 노드 집합로 나누었을 때 매번 두 집합의 경계를 넘는 간선 중 하나를 선택해야 한다.

노드가 주어졌을 때 이 노드의 진출 간선은 선택할 수 있는 후보 간선이다. 이 중 꼭 먼저 처리해야 하는 것은 없다. 어떤 것을 먼저 선택한다고 실제 유리한 것도 없다. 그러면 노드가 주어졌을 때 이 노드의 진출 간선은 모두 후보가 될 수 있으므로 이 간선을 특정 자료구조에 저장한 후에 임의의 하나를 선택하는 형태로 알고리즘을 구현할 수 있다. 이때 꼭 먼저 처리해야 하는 것은 없으므로 저장과 추출은 성능 측면에서 사용하는 자료구조에서 가장 효율적인 방법을 사용할 필요가 있다.

먼저 처리해야 하는 것이 없으므로 리스트, 큐, 스택 등의 사용을 고려해 볼 수 있다. 배열 기반 리스트나 연결 구조 기반 리스트를 생각하면 한쪽 방향으로 삽입하고 추출하는 것이 가장 단순하고 효과적이다. 그런데 이렇게 데이터를 저장하고 추출하는 것은 스택에 해당한다. 스택 대신에 큐를 사용할 수 있다. 하지만 스택, 큐 외에 다른 자료구조의 사용은 스택과 큐와 비교하여 시간적, 공간적 이점이 없다. 스택과 큐를 사용하게 되면 노드를 방문하는 순서가 달라진다. 실제 스택을 사용하는 것이 **깊이 우선**(DFS, Depth-First Search) 탐색 알고리즘이고, 큐를 사용하는 것이 **너비 우선**(BFS, Breadth-First Search) 탐색 알고리즘이다.

너비 우선은 큐를 사용하기 때문에 큐에 먼저 삽입한 노드를 먼저 방문하게 된다. 따라서 출발 노드의 인접 노드를 먼저 방문하게 된다. 깊이 우선은 스택을 사용하기 때문에 특정 경로로 계속 이동하면서 노드를 방문하게 된다. 이때 스택에 노드를 포함하는 순서에 의해 따라가는 경로가 달라진다. 스택을 사용하기 때문에 스택을 사용하지 않고 재귀적으로 구현할 수 있다.

큐에 삽입, 추출하는 비용이나 스택에 삽입, 추출하는 비용은 모두 $O(1)$이다. 이 외에는 두 알고리즘의 차이가 없기 때문에 두 탐색 방법의 성능은 같다. 따라서 어떤 문제가 주어졌을 때 두 탐색 중 어떤 탐색 방법을 사용하여도 해결이 가능한 문제는 보통 너비 우선을 많이 사용한다. 관례적으로 무방향 그래프는 너비 우선을 많이 사용하고 방향 그래프는 깊이 우선을 많이 사용한다.

4. 너비 우선

알고리즘 7.2 너비 우선 탐색 알고리즘

```
1: procedure BFS(G, s)
2:     visited := [false] × |G|
3:     Q := empty Queue of nodes
4:     visited[s] := true
5:     Q.PUSH(s)
6:     while not Q.EMPTY() do
7:         v := Q.POP()
8:         for all e = (v, w) ∈ G do
9:             if not visited[w] then
10:                visited[w] := true
11:                Q.PUSH(w)
```

너비 우선 탐색 알고리즘은 알고리즘 7.2와 같다. 너비 우선 탐색은 일반 그래프 탐색 알고리즘이므로 필요한 용도에 따라 응용하여 사용하게 된다. 보통 방문하면서 노드를 처리해야 할 경우에는 7번째 줄 다음에 큐에서 pop 한 노드 v를 처리하게 된다.

너비 우선 탐색 알고리즘의 성능을 분석하여 보자. 먼저 시간 복잡도를 분석하여 보자. **while** 반복문 이전에 이루어지는 큐의 생성과 **visited** 배열의 생성과 초기화 등의 비용은 최대 $O(n)$으로 분석할 수 있다. **while** 반복문은 매 반복마다 하나의 노드를 처리하며, 총 n개 노드가 있으며, 중복된 노드는 큐에 삽입되지 않으므로 총 n번 반복된다.

내부 **for** 문은 그래프 G를 어떻게 표현하였는지에 따라 다르다. 먼저 반복 횟수를 살펴보자. 인접 행렬이면 매번 n번 반복하므로 이중 반복문의 전체 반복 횟수는 n^2이다. 반면에 인접 리스트이면 내부 반복의 각 반복 횟수는 현재 처리하는 노드에 의해 결정되지만 전체적으로는 모든 간선을 고려하는 형태이다. 더 구체적으로 무방향에서는 각 간선을 정확하게 두 번 고려하게 되며, 방향 그래프에서는 정확하게 한번 고려하게 된다. 따라서 인접 리스트의 경우 이중 반복문의 전체 반복 횟수는 간선 수 m에 비례한다. 반복문 내에서 큐 연산을 제외하면 매 반복에서 일정한 비용만 소요되므로 반복 횟수에 의한 시간 복잡도는 인접 행렬이면 $O(n^2)$, 인접 리스트이면 $O(m)$이다.

반복 과정에서 이루어진 큐 연산은 각 노드마다 한번에 **push**와 **pop**이 필요하다. 따라서 큐와 관련된 전체 비용은 $O(n)$이다. 그러므로 전체 비용은 인접 행렬이면 $O(n) + O(n^2) + O(n) = O(n^2)$이며, 인접 리스트이면 $O(n) + O(m) + O(n) = O(m + n)$이다. 이 비용은 그래프의 방향성 여부와 무관하다. 다중 그래프가 아니고 한 노드에서 다시 그 노드로 가는 간선이 없는 그래프이면 $m \leq n^2$이므로 $O(m + n)$를 $O(n^2)$으로 표현할 수 있지만 $O(m + n)$으로 표현하는 것이 더 많은 정보를 주므로 보통 너비 우선 탐색 알고리즘은 인접 리스트를 사용한다고 가정하고 $O(m + n)$으로 표현하며, 비용은 노드와 간선 수에 비례한다고 말한다.

너비 우선 알고리즘의 공간 복잡도를 분석하여 보자. 입력의 크기는 인접 행렬이면 $O(n^2)$, 인접 리스트이면 $O(m + n)$이다. 내부적으로 **visited** 배열과 큐를 사용하는데, **visited** 배열의 공간 복잡도는 $O(n)$이고, 큐를 최대 사용하였을 때 공간 복잡도도 $O(n)$이다. 실제 완전 그래프이면 모든 노드이 이웃 노드이기 때문에 $n - 1$개 노드가 첫 반복문에서 큐에 삽입된다. 따라서 전체 공간 복잡도는 인접 행렬이면 $O(n^2)$, 인접 리스트이면 $O(m+n)$으로 시간 복잡도와 차이가 없다.

4.1 무방향 그래프에서 최단 경로

BFS를 응용하여 비가중치 그래프에서 출발 노드가 주어졌을 때 이 노드로부터 도달 가능한 모든 노드까지의 최단 경로를 알고리즘 7.3과 같이 구할 수 있다. **distance** 배열의 사용을 제외하고 이 알고리즘과 기본 BFS 알고리즘은 차이가 없다. 실제 **visited** 배열을 사용하지 않고, **distance** 배열만 사용할 수 있다. 이 경우 11번째 줄에서

알고리즘 7.3 너비 우선을 이용한 비가중치 그래프에서 최단 경로 알고리즘

```
1: procedure SHORTESTPATH(G, s)
2:     visited := [false] × |G|
3:     distance := [∞] × |G|
4:     Q := empty FIFO queue of nodes
5:     visited[s] := true
6:     distance[s] := 0
7:     Q.PUSH(s)
8:     while not Q.EMPTY() do
9:         v := Q.POP()
10:        for all e = (v, w) ∈ G do
11:            if not visited[w] then
12:                visited[w] := true
13:                distance[w] := distance[v] + 1
14:                Q.PUSH(w)
```

조건문은 distance = ∞로 바꾸어야 한다.

알고리즘 7.4 BFS를 이용한 연결 여부 확인 알고리즘

```
1: function CONNECTEDCOMPONENTS(G)
2:     ret := empty list of lists
3:     visited := [false] × |G|
4:     for v := 1 to n do
5:         if not visited[v] then
6:             ret.PUSHBACK(BFS(G, visited, v))
7:     return ret
```

모든 연결 부분 그래프도 기본 BFS 알고리즘을 응용하는 알고리즘 7.4을 이용하여 구할 수 있다. 이 알고리즘처럼 연결 그래프가 아닐 때 모든 노드를 탐색해야 하면 너비 우선이나 깊이 우선을 한번만 실행하는 것이 아니라 반복문을 이용하여 아직 방문하지 않은 노드를 출발 노드로 설정하여 여러 번 수행해야 할 수 있다.

5. 깊이 우선

깊이 우선은 스택을 이용하여 구현한다는 것만 차이가 있을 뿐 너비 우선과 알고리즘은 같다. 하지만 깊이 우선은 스택을 이용하기 때문에 재귀적으로 구현할 수 있다. 재귀적 깊이 우선 알고리즘은 알고리즘 7.6과 같다.

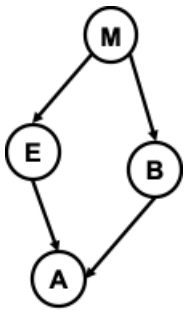
알고리즘 7.5 비재귀적 깊이 우선 탐색 알고리즘

```
1: procedure DFS(G, s)
2:     visited := [false] × |G|
3:     S := empty stack of nodes
4:     visited[s] := true
5:     S.PUSH(s)
6:     while not S.EMPTY() do
7:         v := S.POP()
8:         for all e = (v, w) ∈ G do
9:             if not visited[w] then
10:                visited[w] := true
11:                S.PUSH(w)
```

큐 대신 스택을 사용하는 것을 제외하고 동일하기 때문에 비재귀적 깊이 우선의 성능은 기존 너비 우선과 차이가 없다. 재귀적 깊이 우선도 재귀 호출 비용을 제외하면 성능은 동일하다. 각 노드에 대해 오직 한번의 재귀 호출만 이루어지기 때문에 이 알고리즘에서 일어나는 전체 재귀 호출의 수는 n이다. 또 비재귀적 부분에 일어나는 일은

알고리즘 7.6 비재귀적 깊이 우선 탐색 알고리즘

```
1: procedure DFS(G, s)
2:     visited := [false] × |G|
3:     DFS(G, s, visited)
4: procedure DFS(G, v, visited)
5:     visited[v] := true
6:     for all e = (v, w) ∈ G do
7:         if not visited[w] then
8:             DFS(G, w, visited)
```



<그림 7.4> 위상정렬: M, E, B, A

반복문의 제어와 **visited** 배열과 관련된 작업 뿐이다. 반복문의 반복 횟수는 그래프의 표현에 따라 다르며, 인접 행렬이면 n^2, 인접 리스트이면 m번 반복한다. 따라서 재귀적 깊이 우선의 시간 복잡도도 인접 행렬이면 $O(n^2)$, 인접 리스트이면 $O(m+n)$이다.

비재귀적 깊이 우선의 공간 복잡도도 너비 우선과 차이가 없다. 재귀적 깊이 우선은 스택 자료구조를 사용하지 않지만 재귀 호출 때문에 함수 스택 공간을 사용하게 된다. 이때 출발 노드부터 다른 모든 노드를 지나가는 경로가 존재하면 재귀 호출의 깊이가 n이 되므로 함수 스택을 가장 많이 사용할 경우 공간 복잡도는 $O(n)$이다. 따라서 비재귀적 깊이 우선과 재귀적 깊이 우선의 공간 복잡도는 차이가 없다.

5.1 위상 정렬

위상 정렬(topological sort)은 방향 그래프의 노드를 어떤 순서로 나열하는 것이다. 노드 u에서 v로 간선이 있으면 u는 v보다 이 순서 상에서 앞에 위치해야 한다. 구체적으로 n개 노드로 구성된 방향 그래프 G에서 $f(v)$가 노드 v를 1부터 n으로 중복 없이 매핑하여 주는 함수이면 위상 정렬은 $(u,v) \in G$이면 $f(u) < f(v)$가 되도록 매핑하여 주는 함수 f를 찾는 것을 말한다.

위상 정렬 문제

- 입력. DAG G, $|G| = n$
- 출력. 모든 노드를 1부터 n까지 수로 명명

위상 정렬은 여러 가지 응용에서 사용한다. 예를 들어 대학 교과의 선수이수체계도 위상 정렬에 해당한다. 선후수 관계를 그래프로 표현한 뒤에 위상 정렬을 하면 먼저 이수해야 하는 교과가 앞에 위치하게 된다. 주어진 방향 그래프의 위상 정렬은 독특하지 않을 수 있다. 두 노드는 선후 관계가 없으면 그것의 순서가 바뀌어도 위상 정렬에 해당한다. 그림 7.4에 주어진 방향 그래프에서 $[M,E,B,A]$, $[M,B,E,A]$가 모두 유효한 위상 정렬이다.

모든 방향 그래프를 위상 정렬할 수 있는 것은 아니다. 방향 그래프의 주기가 있으면 위상 정렬을 할 수 없다. 하지만 주기가 없는 모든 방향 그래프는 위상 정렬을 할 수 있다. 이와 같이 주기가 없는 방향 그래프를 DAG(Directed

Acyclic Graph)라 한다. DAG는 항상 소스 노드가 있고 싱크 노드가 있다. 소스 노드는 진입 차수가 0인 노드이다. 반면에 싱크 노드는 진출 차수가 0인 노드이다. 소스 노드와 싱크 노드가 유일하게 존재하는 것은 아니다. 여러 개 존재할 수 있다.

알고리즘 7.7 위상 정렬 알고리즘

```
1: function TOPOLOGICALSORT(G)
2:     label := [0] × |G|
3:     TOPOLOGICALSORT(G, label, 1)
4:     return label
5: procedure TOPOLOGICALSORT(G, label, o)
6:     if |G| = 0 then return
7:     find source node v
8:     label[v] := o
9:     G' := remove v and all (v, u) ∈ G from G
10:    TOPOLOGICALSORT(G', label, o + 1)
```

위상 정렬을 할 때 소스 노드는 앞쪽에 나와야 하며, 싱크 노드는 뒤쪽에 나와야 한다. 소스 노드가 여러 개 있을 때 두 노드의 순서는 중요하지 않다. 싱크 노드가 여러 개 있을 때 싱크 노드 간의 순서도 중요하지 않다. 이 특징을 이용한 위상 정렬 알고리즘은 알고리즘 7.4와 같다. 이 알고리즘을 사용하기 위해서는 그래프가 주어졌을 때 소스 노드를 찾을 수 있어야 한다. 인접 행렬로 그래프를 표현한 경우에는 노드의 열 정보를 통해 소스 노드를 판별할 수 있다. 반면에 인접 리스트는 역인접 리스트가 없으면 소스 노드를 찾는 것이 어렵다. 소스 노드를 찾을 수 있어 이 알고리즘을 적용하더라도 매번 그래프에 대한 수정이 필요하다.

알고리즘 7.8 위상 정렬 알고리즘

```
1: function TOPOLOGICALSORT(G)
2:     order := []
3:     visited := [false] × |G|
4:     for all v in G do
5:         if not visited[v] then
6:             TOPOLOGICALSORT(G, order, visited, v)
7:     label, j := [0] × |G|, n
8:     for all i := 1 to n do
9:         label[order[i]] := j−−
10:    return label
11: procedure TOPOLOGICALSORT(G, order, visited, v)
12:    visited[v] := true
13:    for all w in G do
14:        if v ≠ w and not visited[w] and (v, w) ∈ G then
15:            TOPOLOGICALSORT(G, order, visited, w)
16:    order.PUSHBACK(v)
```

위상 정렬은 DFS를 응용하여 할 수 있다. DFS를 이용하여 탐색하면 소스 노드를 찾기 어렵지만 싱크 노드를 찾기는 쉽다. 싱크 노드는 위상 정렬에서 가장 뒤에 위치해야 하는 노드이다. 이를 응용하면 알고리즘 7.8과 같은 위상 정렬 알고리즘을 만들 수 있다. 제시된 알고리즘에서 알 수 있듯이 이 알고리즘은 재귀적 DFS를 응용하고 있다.

그러면 스택을 이용한 비재귀적 DFS를 응용하여 위상 정렬을 할 수 있을까? 비재귀적으로 DFS를 진행할 경우 싱크를 만나면 이 노드가 싱크인 것은 알 수 있다. 싱크를 만나면 다시 왔던 경로로 거슬러 올라가면서 번호를 할당하면 위상 정렬을 할 수 있는데, 기존 비재귀적 DFS에서는 한번 방문한 노드를 다시 방문하지 않는다. 다시 거슬러 올라가기 위해서는 노드를 두 번 방문하도록 DFS를 변형하여야 하며, 두 번째 방문하였을 때 번호를 할당하면 된다

스택에서 노드를 **pop**하였을 때 그 노드를 바로 다시 스택에 **push**하면 각 노드를 두 번 방문할 수 있으며, 두 번째 방문하는 순서는 탐색하는 순서의 역순이 된다. 문제는 단순하게 노드를 다시 스택에 **push**하면 이 노드를

처음 방문한 것인지 두 번째 방문한 것인지 구분하기 어렵다. 두 번째 방문인지 파악하기 위해 다음 두 가지 방법 중 하나를 사용할 수 있다.

- 방법 1. **visited** 배열을 **boolean** 배열이 아니라 **int** 배열로 바꾸고, 스택에 두 번째로 **push**할 때 값을 2로 설정하여 구분할 수 있다.
- 방법 2. 스택에 두 번째로 **push**할 때 노드 번호의 부호를 바꾸어 **push**하여 구분할 수 있다. 이때 노드 번호가 0인 것이 있으면 $-v-1$을 스택에 **push**한다

알고리즘 7.9 위상 정렬 알고리즘

```
1: function TOPOLOGICALSORT(G)
2:     o := |G|
3:     label := [0] × |G|
4:     visited := [0] × |G|
5:     for all v in G do
6:         if visited[v] = 0 then
7:             Q := empty queue
8:             S := empty stack
9:             visited[v] := 1
10:            S.PUSH(v)
11:            while not S.EMPTY() do
12:                v := S.POP()
13:                if visited[v] = 2 then Q.PUSH(v)
14:                else:
15:                    S.PUSH(v)
16:                    visited[v] := 2
17:                    for all (v, w) in G do
18:                        if visited[w] = 0 then
19:                            visited[w] := 1
20:                            S.PUSH(w)
21:            while not Q.EMPTY() do
22:                v := Q.POP()
23:                label[v] := o − −
24:    return label
```

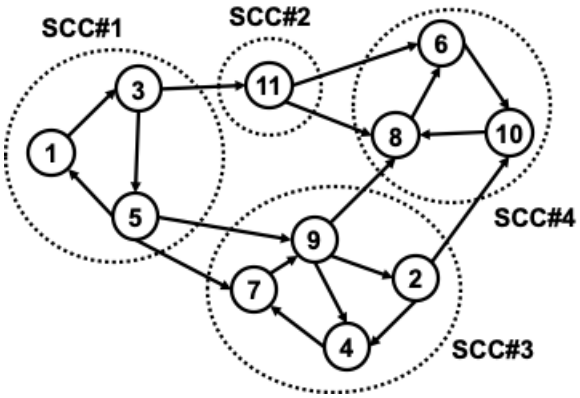
방법 1을 이용한 비재귀적 위상 정렬 알고리즘은 알고리즘 7.9와 같다. 내부 첫 번째 **while** 문이 DFS를 수행하는 부분인데, 13번째부터 16번째 줄을 제외하고는 원 DFS와 차이가 없다. DFS를 수행하면서 스택에서 노드를 pop한 후 두 번째 방문이면 큐에 포함하여, 첫 번째 방문이면 visited 값을 2로 바꾸고 다시 스택에 PUSH한다.

5.2 강한 연결 부분 그래프

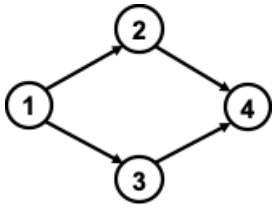
강한 연결 부분 그래프(SCC, Strongly Connect Components)란 방향 그래프 $G = (V, E)$에서 모든 노드 간의 상호 경로가 존재하는 G의 부분 그래프를 말한다. $S \subseteq V$가 SCC이기 위한 필요충분조건은 $u, v \in S$이면 u에서 v, v에서 u로 경로가 모두 존재해야 한다. 방향 그래프에서 SCC를 하나의 노드로 바꾸어 그래프를 다시 표현하면 그림 7.5처럼 SCC로 구성된 DAG로 변환할 수 있다. 방향 그래프에서 SCC를 찾는 것은 무방향 그래프에서 연결 부분 그래프를 찾는 것과 비교되는 문제이다.

그러면 방향 그래프에서 강한 연결 그래프를 어떻게 찾을 수 있을까? DFS를 사용하여 구하는 방법을 생각하여 보자. 그림 7.5의 (1)에서 노드 10, 노드 4, 노드 1에서 DFS를 수행하였을 경우 방문하게 되는 노드는 다음과 같다.

- 노드 10: 10, 8, 6



(1) 방향 그래프의 SCC



(2) SCC를 이용하여 DAG로 변환한 그래프

<그림 7.5> 방향 그래프에서 강한 연결 부분 그래프

- 노드 4: 4, 7, 9, 2, 8, 6, 10
- 노드 1: 전체

여기서 노드 10을 이용하여 탐색하면 SCC를 얻을 수 있지만 다른 노드에서 탐색하면 SCC를 얻을 수 없다. 실제 SCC#1의 경우 어떤 노드에서 탐색을 하더라도 해당 SCC를 얻을 수 없다. 실제 SCC#4를 제외하고 어떤 SCC도 노드의 탐색을 통해 얻을 수 없다. 그러면 SCC#4는 다른 SCC와 어떤 차이가 있어 가능한지 생각하여 보자. 이 SCC는 방향 그래프를 SCC DAG로 변환하였을 때 싱크 노드에 해당한다. 따라서 SCC DAG로 변환하였을 때 싱크 노드에 포함되는 노드에서 탐색을 시작하여 이 SCC를 그래프에서 제거한 다음 이 과정을 반복하면 모든 SCC를 찾을 수 있다. 따라서 방향 그래프에서 SCC DAG로 변환하였을 때 싱크 노드에 포함되는 노드를 찾을 수 있어야 한다.

이 노드를 찾기 위해 이전 절에서 살펴본 위상 정렬을 활용할 수 있다. 우리의 목적은 SCC DAG로 변환하였을 때 싱크 노드에 포함되는 노드를 찾는 것이다. 더 효과적으로 SCC를 찾기 위해서는 DFS의 탐색을 시작할 노드의 순서를 찾는 것이다. 이 순서의 첫 번째 노드에 해당하는 노드에서 DFS를 수행하면 해당 노드가 포함된 SCC를 찾게 되고, 이 SCC에 포함된 노드들을 제외한 다음 그다음 순서에 해당하는 노드에서 DFS를 수행하여 그다음 SCC를 찾게 되는 순서를 확보하면 효과적으로 SCC를 찾을 수 있다.

주어진 방향 그래프에서 위상 정렬을 수행하면 우리가 찾고자 하는 노드가 항상 가장 높은 번호를 부여 받지 않는다. 그런데 위상 정렬의 결과를 보면 SCC DAG에서 소스 노드에 포함되는 노드는 항상 1번을 부여 받는다는 것을 알 수 있다. 이 사실을 어떻게 활용할 수 있을까? 방향 그래프에서 각 간선의 방향을 바꾸어도 SCC는 변하지 않는다. 하지만 SCC DAG로 변환하면 기존 소스 노드는 싱크 노드로 싱크 노드는 소스 노드로 바뀌게 된다. 따라서 모든 간선의 방향을 바꾼 후에 위상 정렬하고, 위상 정렬된 순서로 DFS를 수행하면 모든 SCC를 찾을 수 있다. 간선의 방향을 바꾸어 위상 정렬하기 위해 실제 간선의 방향을 모두 바꿀 필요는 없다. 특히, 인접 행렬이면 행과 열을 바꾸어 위상 정렬을 수행하면 된다.

6. 최단 경로 알고리즘

그래프의 한 노드에서 다른 노드로 가는 최단 경로를 문제는 크게 다음과 같이 분류할 수 있다.

- 종류 1. 한 노드에서 특정 노드까지 최단 경로 찾기
- 종류 2. 한 노드에서 다른 모든 노드까지 최단 경로 찾기 (SSSP: Single-Source Shortest Path)

- 종류 3. 모든 노드에서 다른 모든 노드까지 최단 경로 찾기 (APSP: All-Pairs Shortest Path)

당연하지만 종류 1을 해결하는 알고리즘을 이용하여 종류 2를 해결할 수 있고, 종류 2를 해결하는 알고리즘을 이용하여 종류 3을 해결할 수 있다. 하지만 종류 2와 3은 종류 2와 종류 3 전형 알고리즘으로 해결하는 것이 보통 더 효과적이다.

비가중치 그래프는 DFS, BFS를 이용하여도 최단 경로를 찾을 수 있다. 보통 4.1애서 살펴본 것과 같이 BFS를 이용하여 최단 경로를 찾으며, BFS를 이용한 최단 경로 알고리즘을 다른 말로 Moore 알고리즘이라 한다. 하지만 가중치가 존재하면 단순 그래프 탐색 알고리즘으로 최단 경로를 찾을 수 없다.

가중치 그래프에서 가중치에 음수가 존재하지 않으면 다익스트라(Dijkstra) 알고리즘을 사용하여 최단 경로를 찾을 수 있다. 다익스트라 알고리즘은 SSSP 문제를 해결하여 주는 알고리즘이다. 하지만 음의 가중치가 존재하는 그래프이면 다익스트라 알고리즘을 이용하여 해결할 수 없다. 음의 주기가 없으면 최단 경로는 항상 단순 경로이다. 한 노드를 두 번 지났다는 것은 순환하였다는 것을 의미하며, 이 주기의 가중치 합이 음수가 될 수 없으므로 주기를 제외하면 더 짧은 경로가 된다. 음의 가중치가 존재하는 그래프에서 최단 경로를 찾아주는 알고리즘은 11장에서 살펴본다.

6.1 다익스트라 알고리즘

가중치 방향 그래프에서 SSSP

- 입력. 방향 가중치 그래프 $G = (V, E)$와 출발 노드 s, $e \in E, w(e) \leq 0$
- 출력. 각 노드마다 s부터 해당 노드까지의 최단 경로의 길이

이 절에서는 이 문제를 해결하는 다익스트라 알고리즘을 살펴본다. 이때 출발 노드부터 모든 노드까지 경로가 존재한다고 가정한다. 경로가 존재하지 않아도 문제가 되지 않는다. 먼저 BFS나 DFS를 이용하여 경로가 존재하지 않는 노드를 제거하고 알고리즘을 적용할 수 있다. 보통 방향 가중치 그래프에서 최단 경로 문제를 해결하기 위해 다익스트라 알고리즘을 사용하지만 무방향 가중치 그래프에서도 다익스트라 알고리즘을 이용하여 SSSP 문제를 해결할 수 있다. 모든 간선의 가중치가 같으면 다익스트라 알고리즘을 사용할 필요가 없고 단순 BFS를 통해 해결할 수 있다.

알고리즘 7.10 다익스트라 알고리즘

```
1:  function DIJKSTRA(G, s)
2:      X := {s}
3:      A[s] := 0
4:      B[s] := empty path
5:      while X ≠ V do
6:          min := ∞
7:          for all (v, w) ∈ E and v ∈ X and w ∉ X do
8:              if min > A[v] + D[v][w] then
9:                  min := A[v] + D[v][w]
10:                 (v*, w*) := (v, w)
11:         X.ADD(w*)
12:         A[w*] := A[v*] + D[v*][w*]
13:         B[w*] := B[v*] + (v*, w*)
14:     return A, B
```

다익스트라 알고리즘의 기본 골격은 알고리즘 7.10과 같다. 이 알고리즘의 성능은 **while** 문 내에 **for** 문에 의해 결정된다. 이 알고리즘은 매 순간 가장 최선이 되는 것을 선택하여 알고리즘을 진행하는 대표적인 탐욕적

알고리즘이다. 탐욕적 알고리즘에 대해서는 8장에서 자세히 살펴본다. 다익스트라 알고리즘이 구체적으로 어떻게 매 반복마다 (v^*, w^*)를 선택하는지 조금 뒤로 미루고, 알고리즘의 정확성부터 증명해 보자.

정리 7.2. 다익스트라 알고리즘은 간선의 가중치가 0 이상인 방향 그래프에 대해 출발 노드부터 모든 노드까지의 최단 경로의 길이를 구해준다.

증명 귀납법을 이용하여 다익스트라 알고리즘의 정확성을 증명한다. $L[v]$는 출발 노드 s에서 v까지 실제 최단 경로의 길이이고, $A[v]$는 다익스트라 알고리즘을 통해 구한 s에서 v까지 최단 경로의 길이이면 이 증명은 모든 v에 대해 $A[v] = L[v]$임을 증명해야 한다.

- 귀납 출발: $A[s] = L[s] = 0$. 따라서 귀납 출발은 성립한다.,
- 귀납 가정: 지금까지 완료한 모든 v에 대해 $A[v] = L[v]$임을 가정한다.
- 귀납 절차: 이번 반복에서 (v^*, w^*)를 선택하여 w^*를 X에 추가하였다고 하자. 그러면 다음이 성립한다.

$$A[w^*] = A[v^*] + d[v^*][w^*] = L[v^*] + d[v^*][w^*]$$

우리가 증명해야 하는 것은 $L[v^*] + d[v^*][w^*] = L[w^*]$이다. 이것을 증명하기 위해 s에서 w^*까지 모든 경로의 길이는 $A[w^*]$와 같거나 크다는 것을 증명한다. $A[w^*]$가 s에서 w^*까지 최단 경로이어야 이것이 성립할 수 있다. s에서 w^*까지 임의의 경로 P를 생각하여 보자. s는 X에 있는 노드이고, w^*는 $V - X$에 있는 노드이므로 s에서 w^*까지 임의 경로는 반드시 한번 이상은 X와 $V - X$의 경계를 넘어가야 한다. 따라서 이 경로는 다음 3가지 구간으로 나눌 수 있다.

  - 구간 1. (경계를 최초로 넘기 전까지 구간) X에 있는 노드 y까지의 경로 $\text{path}(s, y)$: 이 경로의 길이는 $A[y]$와 같거나 크다. 참고로 귀납 가정에 의해 $A[y]$는 s에서 y까지 최단 경로이다.
  - 구간 2. (경계를 넘는 최초 간선) y에서 출발하여 $V - X$에 있는 z 노드에서 끝나는 간선: 이 간선의 가중치는 $d[y][z]$이다.
  - 구간 3. (구간 2 이후 w^*까지의 경로) z에서 w^*까지 경로: $\text{path}(s, y) \geq 0$이다.

  이 요소들의 합이 경로 P의 길이가 된다.

$$\text{dist}(P) \geq A[y] + d[y][z]$$

그런데 $A[w^*] = A[v^*] + d[v^*][w^*]$이며 알고리즘 규칙에 의해 $A[y] + d[y][z] \geq A[w^*]$이므로 s에서 w^*까지 모든 경로의 길이는 $A[w^*]$와 같거나 크다.

□

6.2 우선순위 큐 기반 다익스트라 알고리즘

다익스트라 알고리즘은 매번 하나의 노드가 X 집합에 포함되므로 총 바깥 **while** 문은 n번 반복되며, 특별한 자료구조를 사용하지 않으면 내부 **for** 문은 최대 간선 수만큼 검토해야 하므로 전체 시간 복잡도는 $O(mn)$이 된다. 내부 **for** 문은 경계를 건너가는 간선 중 다익스트라 점수가 가장 낮은 것을 찾아야 한다. 여기서 노드 $w \in V - X$의 다익스트라 점수란 모든 $v \in X$에 대해 $A[v] + D[v][w]$가 가장 작은 값을 말한다. 매번 항상 가장 낮은 점수를 찾아야 하므로 이것에 가장 적합한 자료구조는 우선순위 큐이다. 다익스트라 점수는 노드가 기준이므로 각 노드의 다익스트라 점수를 구하여 우선순위 큐에 삽입하면 매번 가장 낮은 다익스트라 점수를 가진 노드를 효율적으로 선택할 수 있다. 우선순위 큐를 이용한 다익스트라 알고리즘은 알고리즘 7.11과 같다.

알고리즘 7.11 우선순위 큐 기반 다익스트라 알고리즘

```
1: function DIJKSTRA(G, s)
2:     X := empty set
3:     A[] := int array of size n
4:     H := empty min heap
5:     for all v ∈ V do
6:         score := 0 if v = s else ∞
7:         H.ADD(score, v)
8:     while not H.EMPTY() do
9:         score, v := H.EXTRACTMIN()
10:        X.ADD(v)
11:        A[v] := score
12:        for all (v, w) ∈ E and w ∈ V − X do
13:            score := H.DELETE(w)
14:            score := MIN(score, A[v] + d[v][w])
15:            H.ADD(score, w)
16:    return A
```

알고리즘 7.11의 시간 복잡도를 분석하여 보자. 여기서 X 집합은 실제 집합 자료구조를 사용할 수 있지만 보통은 **`boolean visited`** 배열을 사용한다. 따라서 X.ADD(v)의 비용은 $O(1)$이다. 보통 우선순위 큐는 삭제 연산을 제공하지 않는다. 하지만 $O(\log n)$ 성능의 삭제 연산을 제공한다고 가정하고 분석을 한다. 위 알고리즘에서 **`while`** 문 이전 비용은 우선순위 큐에 n개 노드를 삽입하는 비용에 의해 좌우되며, 이 비용은 $O(n \log n)$이다. **`while`** 문은 한 번 반복할 때마다 X에 하나의 노드가 추가되므로 총 n번 반복되며, 반복할 때마다 한 번의 우선순위 큐에서 추출 연산을 수행한다. 따라서 이 비용만 고려하면 총 비용은 $n \log n$이다. 내부 **`for`** 문은 복잡하지만 인접 리스트로 그래프를 표현하였다면 방향 그래프이므로 BFS와 마찬가지로 각 간선을 최대 한번 고려하게 된다. 이때 총 두 번의 힙 연산이 이루어지므로 **`while`** 문이 반복하면서 수행되는 전체 **`for`** 문의 시간 복잡도는 $m \log n$이다. 두 비용을 합하면 전체 비용은 $n \log n + m \log n = (m + n) \log n$이다. **`while`** 문 이전 비용을 합하여도 최종 시간 복잡도는 여전히 $(m + n) \log n$이다.

알고리즘 7.12 게으른 다익스트라 알고리즘

```
1: function DIJKSTRA(G, s)
2:     visited := [false] × n
3:     A := [∞] × n
4:     A[s] := 0
5:     H := empty min heap
6:     H.ADD(0, s)
7:     while not H.EMPTY() do
8:         score, v := H.EXTRACTMIN()
9:         if visited[v] then continue
10:        visited[v] := true
11:        A[v] := score
12:        for all (v, w) ∈ E and not visited[w] do
13:            if A[w] < A[v] + d[v][w] then
14:                A[w] := A[v] + d[v][w]
15:                H.ADD(A[w], w)
16:    return A
```

분석에서 언급하였듯이 실제 언어의 라이브러리에서 제공하는 우선순위 큐를 이용하여 다익스트라 알고리즘을 구현하면 우선순위 큐에서 임의 요소를 삭제할 수 없기 때문에 제시된 알고리즘처럼 구현하지 않고, 알고리즘 7.12와 같이 구현하며, 이와 같이 구현하는 다익스트라 알고리즘을 게으른 다익스트라 알고리즘이라 한다.

게으른 다익스트라 알고리즘은 우선순위 큐에 한 노드의 다익스트라 점수가 하나만 존재하는 것이 아니라 여러 개 존재할 수 있다. 따라서 우선순위 큐 연산 비용이 증가하며, 바깥 **`while`** 문의 반복 횟수도 증가할 수 있다. 앞서 분석한 바와 같이 **`while`** 문이 반복하면서 수행되는 전체 **`for`** 문의 반복 횟수는 m에 비례한다. 알고리즘 7.11은

우선순위 큐에 최대 n개 값만 존재하므로 이 비용을 $O(m \log n)$로 분석하였다. 하지만 게으른 다익스트라에서는 우선순위 큐에 한 노드에 대한 여러 다익스트라 점수가 계속 삽입될 수 있다. 따라서 비용은 $O(m \log m)$이 된다. 하지만 간선 수는 최대 n^2개 존재할 수 있으므로 빅O 측면에서 $O(m \log n)$과 $O(m \log m)$은 차이가 없다. 알고리즘 7.11은 **while** 문이 항상 n번 반복하지만 게으른 다익스트라에서는 최대 m번 반복할 수 있다. 따라서 8번째 줄에서 **extractMax**에 의한 비용은 $O(m \log m)$이다. 따라서 **while** 문 전체 비용은 $O(m \log m) = O(m \log n)$이다. **while** 문 이전 비용을 합하여도 시간 복잡도는 바뀌지 않는다. 결과적으로 게으른 다익스트라 알고리즘의 시간 복잡도는 7.11 알고리즘과 같은 수준이다.

퀴즈

1. 너비 우선과 깊이 우선 탐색은 자료구조로 큐와 스택을 사용하는 것만 차이가 있을 뿐 동일하게 구현할 수 있다. 인접 리스트를 이용하여 구현하였다고 가정하고, 두 알고리즘의 시간 복잡도를 분석하고자 한다. 둘 다 VISITED 배열을 초기화하는 비용 $O(n)$이 필요하다. 그다음 각 큐, 스택이 비어 있을 때까지 반복문을 수행하게 된다. 각 반복마다 두 알고리즘은 하나의 노드를 자료구조에서 추출한 다음 이것의 인접 노드 중 방문하지 않는 노드를 다시 자료구조에 삽입하게 된다. 따라서 반복문의 전체 비용은 큐 또는 스택 연산 비용과 인접 노드 탐색 비용으로 나눌 수 있다. 다음 중 큐 또는 스택 연산의 총 비용과 인접 노드 탐색의 총 비용을 올바르게 나타낸 것은?

   ① $O(n)$, $O(m)$
   ② $O(m)$, $O(n)$
   ③ $O(n^2)$, $O(m)$
   ④ $O(n)$, $O(n^2)$

2. 깊이 우선 탐색은 재귀 호출을 이용하여 구현할 수 있다. 다음 중 이것의 이점으로 보기 힘든 것은?

   ① 스택 자료구조를 직접 생성하여 사용하지 않아도 된다.
   ② 노드의 방문 여부를 알기 위한 배열이 필요 없다.
   ③ 자연스럽게 한 쪽 방향으로 탐색할 수 있을 때까지 탐색한다.
   ④ 재귀 호출을 할 경우 함수 스택 공간이 필요하기 때문에 최악의 경우 사용하는 스택 공간의 공간 복잡도는 $O(n)$이다.

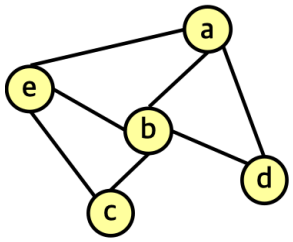
3. 너비 우선 탐색을 이용하여 방향 그래프를 탐색하고자 한다. 그래프를 인접 행렬 또는 인접 리스트로 표현하였을 때에 따라 인접 노드를 찾는 방법이 달라진다. 인접 리스트를 사용할 경우 탐색 과정에서 각 간선을 몇 번 활용하는가?

   ① 2
   ② 알 수 없음
   ③ 1
   ④ 0

4. 깊이 우선 탐색의 경우 스택에 삽입할 때 스택에 이미 삽입한 노드는 다시 삽입하지 않기 위해 삽입하기 전에 스택에 삽입된지 여부를 검사할 수 있다. 하지만 이 경우 한 노드에서 한 방향으로 계속 탐색을 못하게 된다. 다음 알고리즘을 사용한다고 가정하였을 때,

```
1: visited[i] := [false] × n
2: s := starting node
3: S := empty stack of nodes
4: S.PUSH(s)
5: while S is not empty do
6:     v := S.POP()
7:     if visited[v] then continue
8:     else
9:         visited[v] := true
10:        for all e(v, w) ∈ G do
11:            S.PUSH(w)
```

다음 그래프에서 a를 시작 노드로 설정하고 위 알고리즘을 적용하였을 때 방문하는 노드의 순서는?



가정. `visited`$[v]$를 **true**로 설정한 순간이 해당 노드를 방문한 순서라 하고, 스택에 삽입할 때 이웃 중 노드 이름이 큰 것을 먼저 스택에 삽입함

① a, b, c, e, d

② a, b, d, c, e

③ a, e, c, b, d

④ a. e, c, d, b

연습문제

1. $m \times n$ 그리드가 주어진다. 각 셀에 0 이상의 수가 주어진다. 가장 왼쪽 위 셀에서 가장 오른쪽 아래 셀까지 가는 경로 중 합계가 가장 작은 경로를 찾아라. 이동은 한 번에 오른쪽 또는 아래 셀로 이동할 수 있다. 예를 들어 3×3 셀이 다음과 같이 주어지면 합이 최소인 경로는 $1 \to 3 \to 1 \to 1 \to 1$이고 그것의 합은 7이다.

$$\begin{matrix} 1 & 3 & 1 \\ 1 & 5 & 1 \\ 4 & 2 & 1 \end{matrix}$$

2. 이진 트리의 루트 노드가 주어졌을 때 이 트리의 지름을 구하라. 이진 트리의 구조체는 다음과 같이 정의한다.

```
TreeNode:
    val: int
    left: TreeNode
    right: TreeNode
```

   트리의 지름은 한 노드에서 다른 노드까지 가장 긴 경로의 길이를 말하며, 지름에 해당하는 경로는 루트를 지나갈 수 있고 지나가지 않을 수 있다.

3. 알고리즘 7.11에 제시된 이진 힙을 이용한 다익스트라 알고리즘은 힙에서 데이터를 삭제하는 연산을 사용하고 있으며, 그것의 비용을 $O(\log n)$으로 가정하고 분석하고 있다. 이 연산을 어떻게 구현할 수 있는지 제시하시오. 보통 프로그래밍 언어 라이브러리에서 제공하는 우선순위 큐는 이와 같은 연산을 제공하지 않는다. 그 이유를 설명하시오.