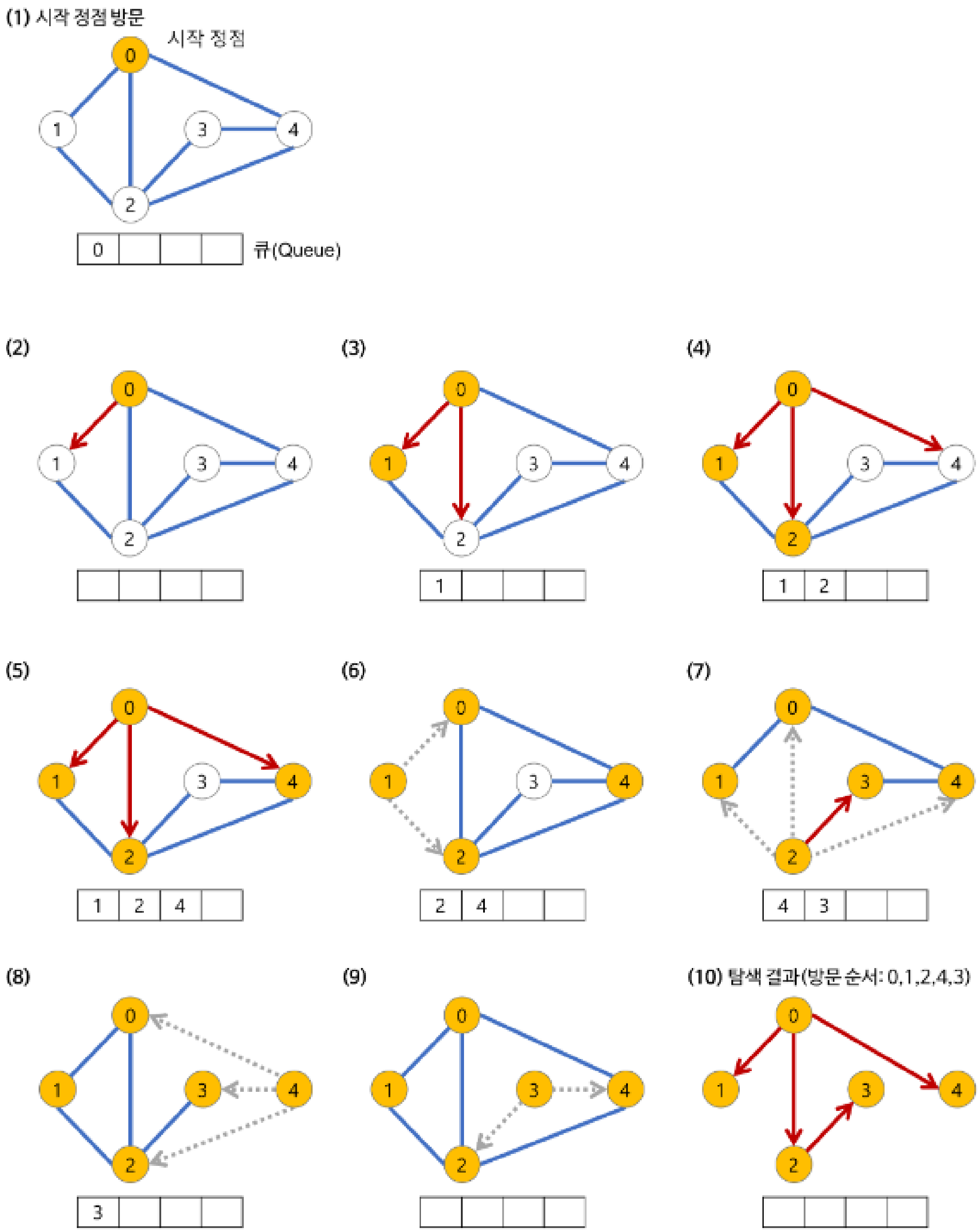
<3주차> BFS와 DFS

❓ 들어가기에 앞서, 그래프 탐색의 개념을 알아야한다.

→ 어떤 한 그래프와 해당 그래프의 시작 정점이 주어졌을때, 시작점에서 **간선(Edge, E)**을 타고 이동할 수 있는 **정점(Vertex, V)**들을 모두 찾아야 하는 문제를 의미
→ Directed와 Undirected 모두 적용
→ 이를 해결하는 알고리즘이 BFS와 DFS

BFS

- 시작 노드에서 인접한 노드들을 먼저 탐색 후, 차례대로 더 멀리 있는 노드들을 탐색하는 방식
- BFS는 방문한 노드들을 차례로 저장한 후 꺼낼 수 있는 큐(Queue) 자료구조를 사용하여 구현.
- 주로 최단 경로를 찾을 때 유용.
- 재귀적으로 동작하지 않음.
- **방문 여부 확인 필수!!**



1. 시작 노드를 방문한다. (방문한 노드 체크)
2. 큐에 방문된 노드를 삽입(enqueue)한다. (초기 상태의 큐에는 시작 노드만이 저장)
3. 큐에서 노드를 하나 꺼내고 그 노드와 인접한 노드들을 모두 방문한다. (인접한 노드가 없다면 큐의 앞에서 노드를 꺼낸다(dequeue))
4. 큐에 방문된 노드를 삽입(enqueue)한다.

5. 2번부터 4번과정을 큐가 소진될 때까지 계속한다.

코드 구현

```
from collections import deque

def bfs(graph, start):
    visited = set()  # 방문한 노드를 저장할 집합
    queue = deque([start])  # 탐색할 노드를 저장할 큐

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            queue.extend(graph[node] - visited)

graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F', 'G'},
    'D': {'B'},
    'E': {'B', 'H'},
    'F': {'C'},
    'G': {'C'},
    'H': {'E'}
}

bfs(graph, 'A')  # 시작 노드 A라고 가정.
```

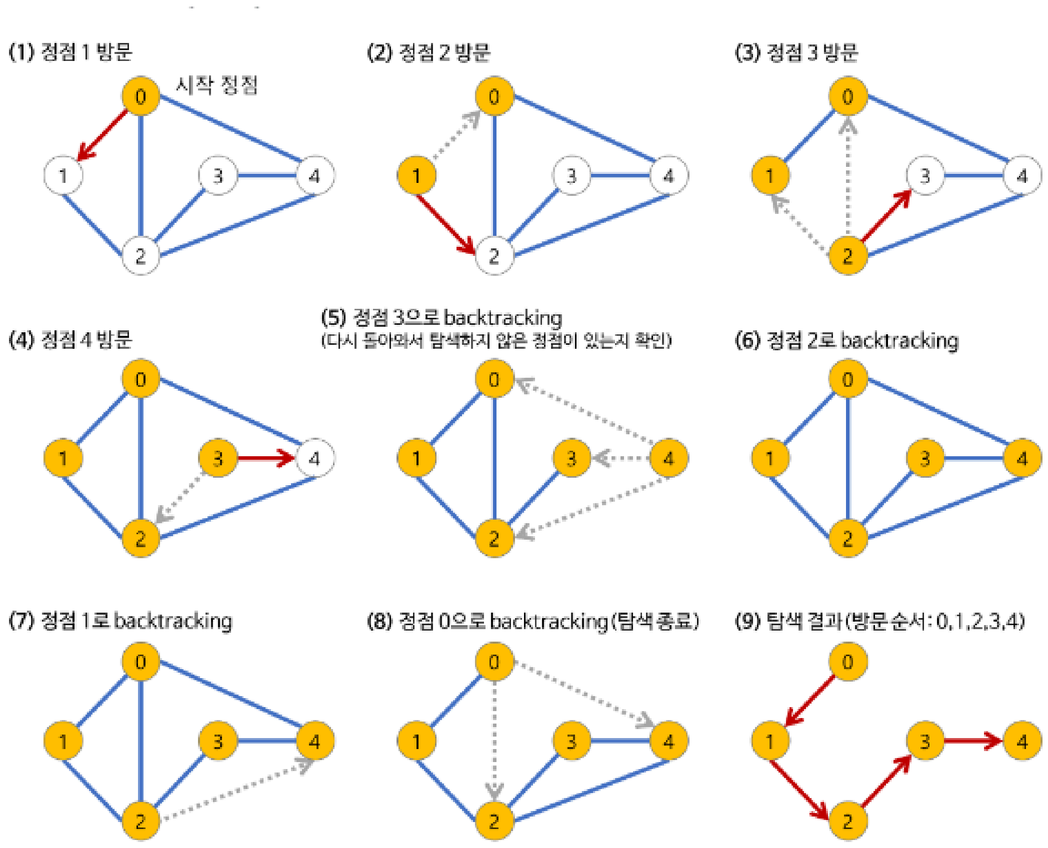
BFS 특징

- 큐(Queue) 자료구조를 사용하는 것에서 알 수 있듯이, 선입선출의 원칙
- 인접 리스트로 표현된 그래프의 시간복잡도: O(N+E)
- 인접 행렬로 표현된 그래프의 시간복잡도: O(N^2)

→ 따라서 그래프 내에 적은 숫자의 간선만을 가지는 희소 그래프(Sparse Graph) 의 경우 인접 행렬보다 인접 리스트를 사용하는 것이 유리하다.

DFS

- 루트 노드에서 시작해서 다음 분기(branch)로 넘어가기 전에 해당 분기를 완벽하게 탐색하는 방법.
- 재귀호출 혹은 스택(stack) 자료구조를 사용하여 구현.
- 주로 모든 노드를 방문하거나 특정 경로를 찾기 위해 사용.
- 순환 알고리즘의 형태.
- **방문 여부 확인 필수!!**



1. 시작 노드를 방문한다. (방문한 노드 체크)
2. 시작 노드와 인접한 노드들을 차례로 순회한다. (인접한 노드가 없다면 종료)
3. 시작 노드와 이웃한 노드 a를 방문했다면, 시작 노드와 인접한 또 다른 노드를 방문하기 전에 a의 이웃한 노드들을 전부 방문해야한다.

4. a를 시작 정점으로 DFS를 다시 시작하여 a의 이웃 노드들을 방문한다
5. a의 분기를 전부 완벽하게 탐색했다면 다시 시작 노드에 인접한 정점들 중에서 아직 방문이 안 된 정점을 찾는다.

   → 즉, a의 분기를 전부 완벽하게 탐색한 뒤에야 시작 노드의 다른 이웃 노드를 방문할 수 있다는 뜻.
6. 방문이 안 된 정점이 있으면 다시 그 정점을 시작 정점으로 DFS를 하며 없다면 종료한다.

코드 구현 (재귀)

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=" ")

    for next_node in graph[start] - visited:
        dfs(graph, next_node, visited)

graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F', 'G'},
    'D': {'B'},
    'E': {'B', 'H'},
    'F': {'C'},
    'G': {'C'},
    'H': {'E'}
}

dfs(graph, 'A') # 시작 노드 A라고 가정.
```

코드 구현 (스택)

```
def dfs_stack(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(node, end=" ")
            # 스택에 인접 노드를 추가 (역순으로 추가하여 올바른 순서로
            stack.extend(graph[node] - visited)

graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F', 'G'},
    'D': {'B'},
    'E': {'B', 'H'},
    'F': {'C'},
    'G': {'C'},
    'H': 출ㅓ
```

DFS 특징

- 스택(stack) 자료구조 혹은 재귀의 형식으로 구현 가능
- 인접 리스트로 표현된 그래프의 시간복잡도: O(N+E)
- 인접 행렬로 표현된 그래프의 시간복잡도: O(N^2)

출처

https://gmlwjd9405.github.io/2018/08/15/algorithm-bfs.html

https://gmlwjd9405.github.io/2018/08/14/algorithm-dfs.html