알고리즘 스터디

ToBig's 11기 이영전

Ch12. 그래프(Graph)

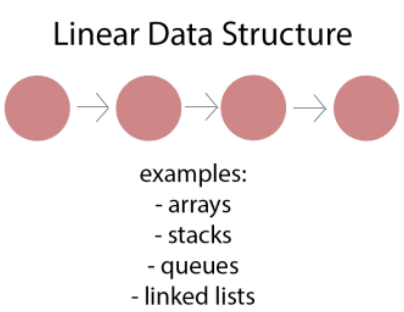
- 비선형 자료구조
- Graph
- DFS. BFS
- 백트래킹, 제약충종 문제
- 문제해설
- 문제 풀이!

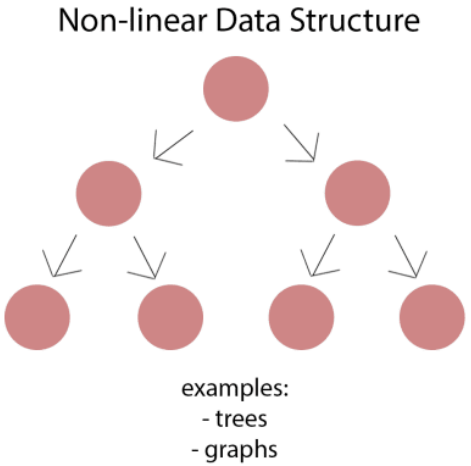
Contents

Unit 01 | 비선형 자료구조

- **비선형 자료구조**

- 데이터 요소가 순차적(Sequential)으로 또는 선형으로 배열되지 않는 자료구조를 비선형(Non-Linear) 자료구조라고 한다
  - 비선형 자료구조는 선형과 달리 멀티 레벨로 구성
  - 대표적인 비선형 자료구조(Non-linear Data Structure)로 Graph와 Tree가 있다
  - 탐색이 복잡하고 선형에 비해 구현도 다소 번잡한 편이지만, 메모리를 좀 더 효율적인 장점이 있다





Unit 02 | Graph Data Structure

"Node와 Edge로 항목들의 관계를 그림으로 표현한 것이 그래프이다"

- Graph란?
  - 그래프 이론에서 그래프란 객체의 일부 쌍들이 '연관되어' 있는 객체 집합 구조를 말한다

- Graph 이론의 배경
  - '수학의 모차르트' 레온하르트 오일러의 조사
  - 도형 문제처럼 보이지만, 당시까지 알려진 기하학으로는 풀 수 없음을 깨달음
  - 쾨니헤스베르크의 프레겔 강 다리 문제를 풀면서 그래프 이론을 창시

Unit 02 | Graph Data Structure

오일러 경로

- 오일러 경로

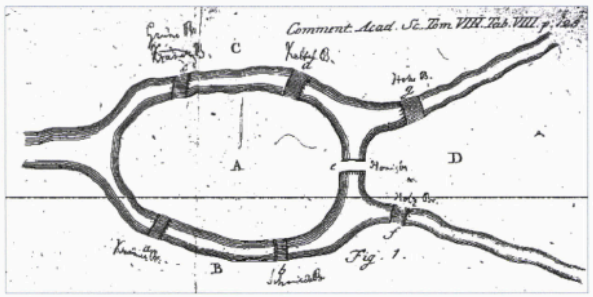


그림 12-2 18세기 오일러가 논문에서 직접 그린 쾨니히스베르크 다리 스케치[4]

- 논문에 포함된 이 스케치는 현대에 이르러 '그래프 구조'의 원형이 되었다

- 현대적인 문제 풀이
  - Node와 Edge(또는 Vertex와 Edge)

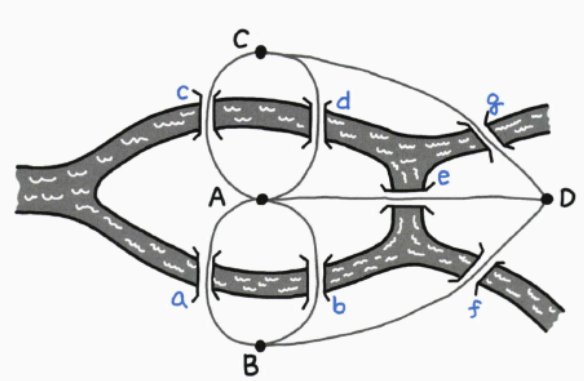
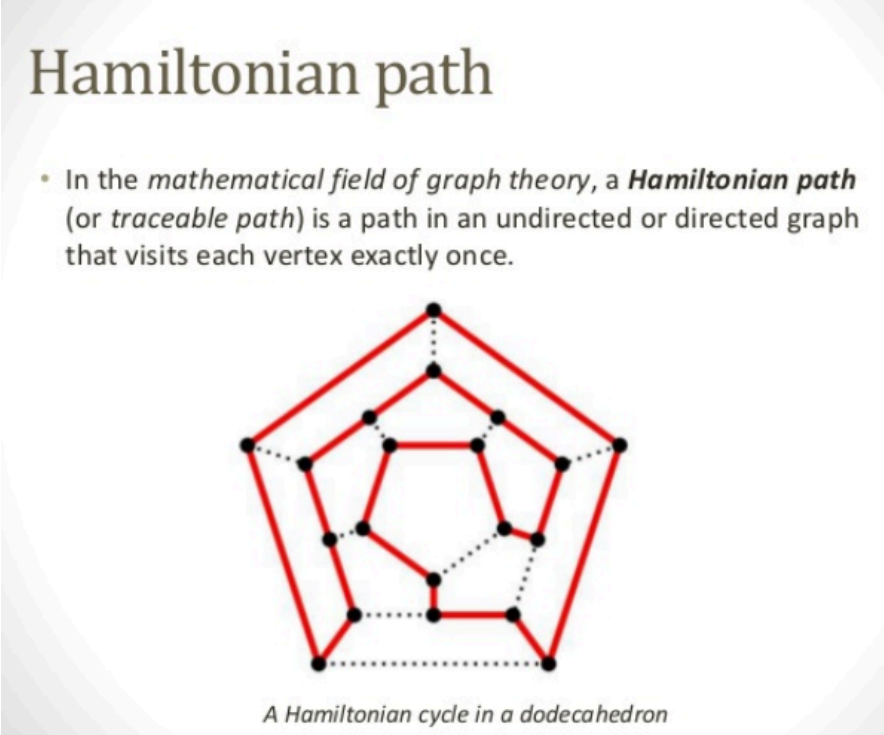


그림 12-3 쾨니히스베르크 다리의 그래프 구조[5]

- 오일러 정리
  - "모든 정점(Node, Vertex)가 짝수 개의 차수(Degree)를 갖는다면 모든 다리를 한 번씩만 건너서 도달하는 것이 성립한다" -오일러-
  - 그로부터 100년이 훨씬 더 지난 1873년, 독일의 수학자 칸 히어홑저가 수학적으로 증명
  - 이를 오일러의 정리(Euler's Theorem)이라고 한다
- 오일러 경로
  - a.k.a 한붓그리기
  - 모든 간선(Edge)을 한 번씩 방문하는 유한 그래프(Finite Graph)를 일컬어 오일러 경로(Euler Trail/Eulerian Path)라고 부른다

Unit 02 | Graph Data Structure



- 해밀턴 경로
  - 해밀턴 경로는 각 정점을 한 번씩 방문하는 무향 또는 유향 그래프 경로를 말한다
  - Q. 해밀턴 경로(Hamiltonian Path)와 오일러 경로의 차이점은 무엇일까?
  - 오일러 경로, 해밀턴 경로
    - 오일러 경로: Edge를 기준으로 한다
    - 해밀턴 경로: Node를 기준으로 한다
  - 그러나 이러한 단순한 차이에도 불구하고 해밀턴 경로를 찾는 문제는 최적 알고리즘이 없는 대표적인 NP-완전(Complete) 문제다

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

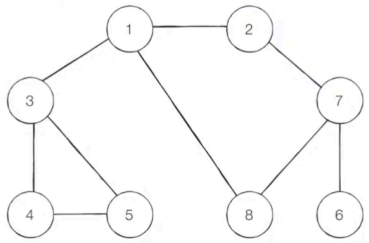
DFS

- Depth-First-Search
  - 깊이 우선 탐색
  - 그래프에서 깊은 부분을 우선적으로 탐구하는 알고리즘입니다
- DFS 동작 과정은 다음과 같습니다
  - 탐색의 시작노드를 스택에 삽입하고 방문 처리를 한다
  - 스택의 최상단 노드에 방문하지 않은 인접노드가 하나라도 있다면 그 노드를 스택에 넣고 방문 처리 한다방문하지 않은 인접 노드가 없으면 스택에서 최상단 노드를 꺼낸다
  - 더이상 2번의 과정을 수행할 수 없을 때까지 반복한다

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

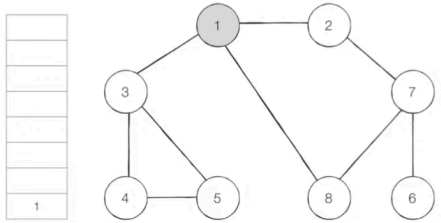
DFS 동작 예시

- [Step 0] 그래프를 준비합니다. (**방문 기준**: 번호가 낮은 인접 노드부터)
  - 시작 노드: 1



DFS 동작 예시

- [Step 1] 시작 노드인 '1'을 스택에 삽입하고 방문 처리를 합니다.



DFS 소스코드 예제(Python)

In [ ]:
```python
# Do-it
# Queue(Python의 경우 deque) 자료구조를 활용해서 DFS 알고리즘을 구현해보자

# graph: 각 노드가 연결된 정보 표현(2차원 리스트)
graph = [
         [],
         [2, 3, 8],
         [1, 7],
         [1, 4, 5],
         [3, 5],
         [3, 4],
         [7],
         [2, 6, 8],
         [1, 7]
]

# 각 노드가 방문된 정보를 표현(1차원 리스트)
visited = [False] * 9
```

In [ ]:
```python
from collections import deque

def bfs(graph, start, visited):
    # 시작노드를 큐에 추가하고 방문처리한다
    queue = deque([start])  # deque = [start]
    visited[start] = True

    while queue:  # 큐가 빌때까지 반복한다
        # 시작노드를 queue에서 제거한다
        v = queue.popleft()
        print(v, end = ' ')

        # 인접노드가 있으면 큐에 추가한다
        for i in graph[v]:
            if not visited[i]:  # 방문한 적이 없는 노드를 추가해준다
                queue.append(i)
                visited[i] = True
```

In [ ]:
```python
bfs(graph, 1, visited)
```

```
1 2 3 8 7 4 5 6
```

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

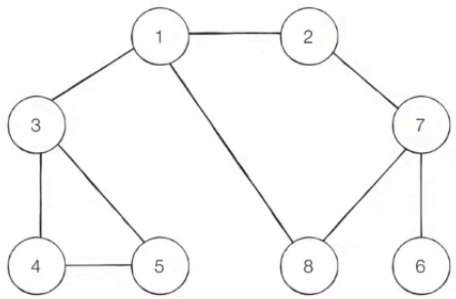
BFS

- BFS는 너비 우선 탐색이라고도 부르며, 그래프에서 가까운 노드부터 우선적으로 탐색하는 알고리즘이다
- BFS는 큐 자료구조를 이용하며, 구체적인 동작 과정은 다음과 같다
  - 1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 한다
  - 1. 큐에서 노드를 꺼낸 뒤에 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리를 한다
  - 1. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복한다

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

BFS 동작 예시

- [Step 0] 그래프를 준비합니다. (**방문 기준**: 번호가 낮은 인접 노드부터)
  - 시작 노드: 1



DFS 소스코드 예제(Python)

In [ ]:
```python
# Do-it
# Queue(Python의 경우 deque) 자료구조를 활용해서 DFS 알고리즘을 구현해보자

# graph: 각 노드가 연결된 정보 표현(2차원 리스트)
graph = [
         [],
         [2, 3, 8],
         [1, 7],
         [1, 4, 5],
         [3, 5],
         [3, 4],
         [7],
         [2, 6, 8],
         [1, 7]
]

# 각 노드가 방문된 정보를 표현(1차원 리스트)
visited = [False] * 9
```

In [ ]:
```python
from collections import deque

def bfs(graph, start, visited):
    # 시작노드를 큐에 추가하고 방문처리한다
    queue = deque([start])  # deque = [start]
    visited[start] = True

    while queue:  # 큐가 빌때까지 반복한다
        # 시작노드를 queue에서 제거한다
        v = queue.popleft()
        print(v, end = ' ')

        # 인접노드가 있으면 큐에 추가한다
        for i in graph[v]:
            if not visited[i]:  # 방문한 적이 없는 노드를 추가해준다
                queue.append(i)
                visited[i] = True
```

In [ ]:
```python
bfs(graph, 1, visited)
```

```
1 2 3 8 7 4 5 6
```

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

- DFS 구현
  - DFS는 주로 스택으로 구현하거나 재귀로 구현하며, 이후에 살펴볼 백트래킹을 통해 뛰어난 효용을 보인다
- BFS 구현
  - 반면 BFS는 주로 큐로 구현하며, 그래프의 최단 경로를 구하는 문제 등에 사용된다
  - 13장에서 40번 '네트워크 딜레이 타임'문제를 통해 다익스트라 알고리즘으로 최단 경로를 찾는 문제에서 BFS로 구현하는 코드를 살펴볼 것이다

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

백트래킹

- 백트래킹(Backtracking)은 해결책에 대한 후보를 구축해 나아가다 가능성이 없다고 판단되는 즉시 후보를 포기(backtrack)해 정답을 찾아가는 범용적인 알고리즘
- 제약 충족 문제(Constraint Satisfaction Problem)에 특히 유용함

- Background
  - DFS(깊이 우선 탐색)를 이야기하다 보면 항상 백트래킹이라는 단어가 함께 나오며 백트래킹은 DFS보다 좀 더 광의의 의미를 지닌다
  - 백트래킹은 탐색을 하다가 더 갈 수 없으면 왔던 길을 되돌아가 다른 길을 찾는다는 데서 유래했다
  - 백트래킹은 DFS와 같은 방식으로 탐색하는 모든 방법을 뜻하며, DFS는 백트래킹의 골격을 이루는 알고리즘이다.

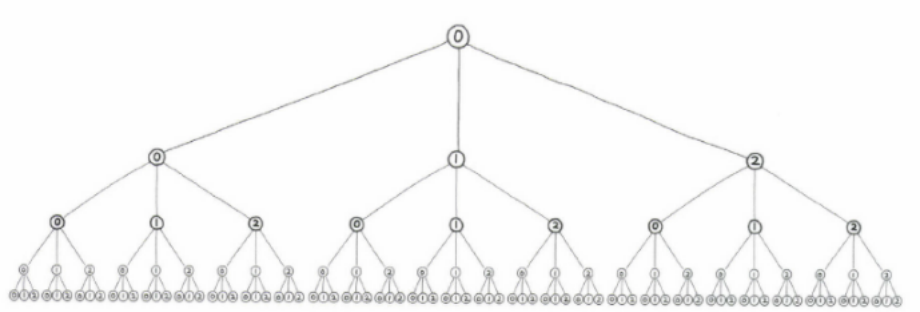


그림 12-9 탐색해야 하는 가지가 많은 트리

트리 가지치기(Pruning)

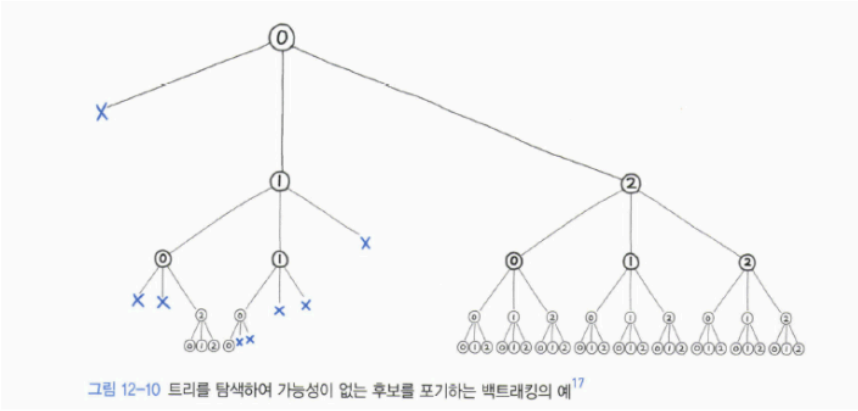


그림 12-10 트리를 탐색하여 가능성이 없는 후보를 포기하는 백트래킹의 예[17]

- DFS로 탐색을 시도하고, 가능성이 없는 후보는 즉시 포기하고 백트래킹한다면
- 트리의 불필요한 거의 대부분을 버릴 수 있다
- 이처럼 불필요한 부분을 일찍 포기한다면 탐색을 최적화할 수 있기 때문에, 가지치기는 트리의 탐색 최적화 문제와도 관련이 있다.

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

백트래킹

- 백트래킹의 구현
  - 주로 재귀로 구현하며, 알고리즘마다 DFS 변형이 조금씩 일어나지만 기본적으로 DFS의 범주에 속한다
  - 백트래킹은 가보고 되돌아오고를 반복한다. 운이 좋으면 시행착오를 덜 거치고 목적지에 도착할 수 있지만 최악의 경우에는 모든 경우를 거친 다음에야 도착할 수 있다. 이 때문에 브루트 포스와 유사하다
  - 하지만 한 번 방문 후 가능성이 없는 경우에는 즉시 후보를 포기한다는 점에서 매번 같은 경로를 방문하는 브루트 포스보다는 훨씬 더 우아한 방식이라 할 수 있다

Unit 03 | DFS, BFS, 백트래킹, 제약 충족 문제

제약 충족 문제

- 제약 충족 문제란
  - 수많은 제약 조건(Constraint)을 충족하는 상태(State)를 찾아내는 수학 문제를 일컫는다
  - 백트래킹은 제약 충족 문제(이하 CSP)를 풀이하는 데 필수적인 알고리즘이다. 앞서 살펴본 가지치기를 통해 제약 충족문제를 최적화 하기 때문이다

- 제약 충족 문제는 인공지능이나 경영 과학 분야에서 심도 있게 연구되고 있으며, 합리적인 시간 내에 문제를 풀기 위해 휴리스틱과 조합 탐색 같은 개념을 함께 결합해 문제를 풀이한다.

	9	4				1	3	
				7	6			2
	8			1				
	3	2						
			2				6	
				5		4		
					8			7
		6	3		4			8

- 대표적인 제약 충족 문제로는 Sudoku가 있다

Unit 04 | Q33. 전화 번호 문자 조합

```python
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        """
        :type digits: str
        :rtype: List[str]
        """

        if digits == "":  # if there aren't any strings, return []
            return []

        nums_to_letters = {
            2: "abc",
            3: "def",
            4: "ghi",
            5: "jkl",
            6: "mno",
            7: "pqrs",
            8: "tuv",
            9: "wxyz"
        }

        combs = [""]  # combination

        for digit in digits:
            new_combs = []
            for comb in combs:
                for letter in nums_to_letters[int(digit)]:
                    new_combs.append(comb + letter)
            combs = new_combs  # update combs

        return combs
```

Unit 05 | Q39. 코드 스케줄

들어주셔서 감사합니다.