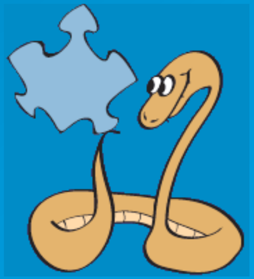
파이썬 알고리즘

ALGORITHMS IN PYTHON

최영규 지음



CHAPTER 03

억지기법과 완전 탐색

학습 내용

3.1 선택 정렬

3.2 순차 탐색

3.3 문자열 매칭

3.4 최근접 쌍의 거리

3.5 완전 탐색(Exhaustive search)

3.6 그래프 탐색

억지 기법(brute-force)

- 문제의 정의를 바탕으로 한 가장 직접적인 방법
  - brute-force method(억지 기법)
  - Naïve method(단순, 순진한 방법)
  - 예: 최대 공약수 방법1, a^n을 구하는 알고리즘 등
- 억지 기법의 중요성

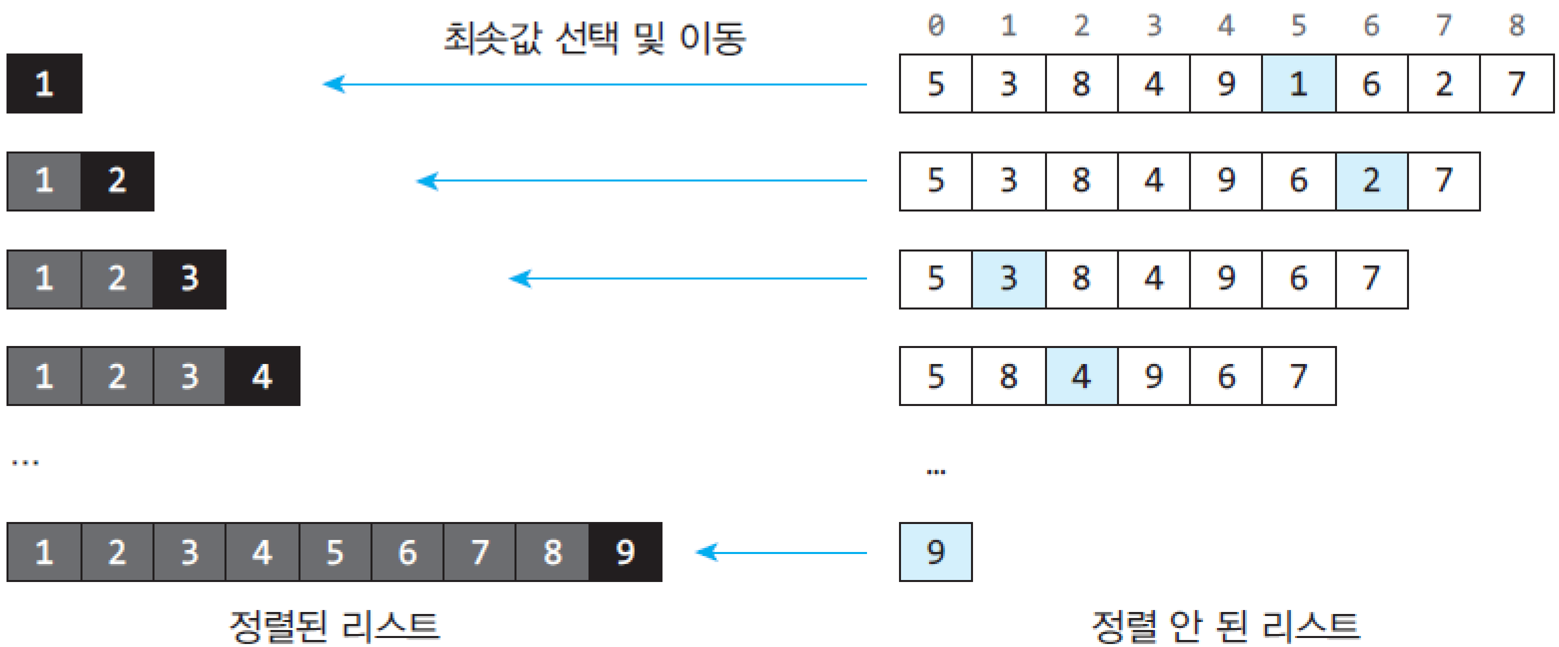
> - 해결하지 못하는 것보다는 단순하게라도 해결하는 것이 훨씬 좋다. 또한, 쉬운 문제를 어렵게 풀 필요는 없다.
> - 억지 기법은 매우 광범위한 문제에 적용할 수 있는 알고리즘 설계 기법이다.
> - 입력의 크기가 작은 경우 억지 기법이 충분히 빠를 수 있고, 심지어 점근적으로 더 효율적인 알고리즘보다 실제로는 더 빠를 수도 있다.
> - 더 효율적인 알고리즘의 설계와 분석을 위한 이론적인 기반이 된다.

- 정렬, 탐색, 기하학적 문제, 완전 탐색, 그래프 탐색

3.1 선택 정렬

- 정렬 문제에 대한 가장 직접적인 해결 방법? **억지기법**

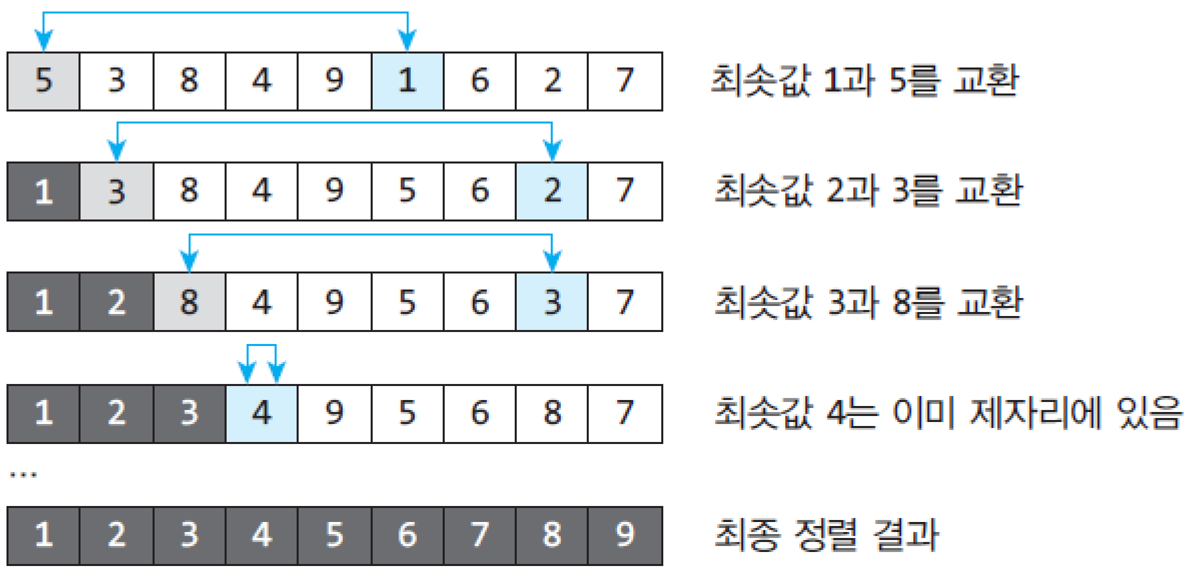
> 입력 리스트에서 가장 작은 항목을 찾고, 이것을 꺼내 정렬된 리스트에 순서대로 저장한다.



추가적인 리스트 필요: 저장공간 필요하지 않도록 알고리즘 개선

제자리 정렬을 위한 알고리즘 개선

정렬이 안 된 리스트에서 최솟값이 선택되면 이 값을 새로운 리스트에 저장하는 것이 아니라 **첫 번째 요소와 교환**한다.



알고리즘

알고리즘 3.1 선택 정렬

```
def selection_sort(A) :
    n = len(A)
    for i in range(n-1) :
        least = i
        for j in range(i+1, n) :
            if (A[j]<A[least]) :
                least = j
        A[i], A[least] = A[least], A[i]
        printStep(A, i + 1);
```

알고리즘 테스트 선택 정렬

```
data = [ 5, 3, 8, 4, 9, 1, 6, 2, 7 ]
print("Original  : ", data)
selection_sort(data)
print("Selection : ", data)
```

C:\WINDOWS\system32\cmd.exe

```
Original  : [5, 3, 8, 4, 9, 1, 6, 2, 7]
  Step  1 = [1, 3, 8, 4, 9, 5, 6, 2, 7]
  Step  2 = [1, 2, 8, 4, 9, 5, 6, 3, 7]
  Step  3 = [1, 2, 3, 4, 9, 5, 6, 8, 7]
  Step  4 = [1, 2, 3, 4, 9, 5, 6, 8, 7]
  Step  5 = [1, 2, 3, 4, 5, 9, 6, 8, 7]
  Step  6 = [1, 2, 3, 4, 5, 6, 9, 8, 7]
  Step  7 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
  Step  8 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
Selection : [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- 무엇이 입력의 크기를 나타내는지를 먼저 명확히 결정

복잡도 분석

- 입력의 크기: 리스트의 전체 항목의 수 *n*
- 기본 연산: 비교 연산 A[j]<A[least]
- 입력 구성에 따른 차이:
  – 입력에 상관없이 항상 일정한 횟수의 비교 연산이 필요

- 복잡도

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{n(n-1)}{2} \in O(n^2)$$

3.2 순차 탐색

문제 3.2 정렬되지 않은 리스트의 탐색 문제

> 리스트에 n개의 항목이 들어 있다. 이 리스트에서 "탐색키"를 가진 항목을 찾아라. 만약 찾는 항목이 있으면 그 항목의 인덱스를 반환하고, 없으면 −1을 반환하라. 단, 리스트의 항목들은 정렬되어 있지 않다.

- 억지 기법 탐색 전략
  - 순차 탐색(sequential search) 또는 선형 탐색(linear search)

4를 찾는 경우	0	1	2	3	4	5
4 ≠ 5 계속 탐색	5	3	8	4	2	7
4 ≠ 3 계속 탐색	5	3	8	4	2	7
4 ≠ 8 계속 탐색	5	3	8	4	2	7
4 = 4 탐색 성공	5	3	8	4	2	7

인덱스 3 반환

6을 찾는 경우	0	1	2	3	4	5
6 ≠ 5 계속 탐색	5	3	8	4	2	7
6 ≠ 3 계속 탐색	5	3	8	4	2	7
6 ≠ 8 계속 탐색	5	3	8	4	2	7
6 ≠ 4 계속 탐색	5	3	8	4	2	7
6 ≠ 2 계속 탐색	5	3	8	4	2	7
6 ≠ 7 계속 탐색	5	3	8	4	2	7

탐색 실패
-1 반환

알고리즘과 복잡도

알고리즘 3.2 순차 탐색(다시 보기)

```
def sequential_search(A, key):
    for i in range(len(A)) :
        if A[i]  ==  key :
            return i
    return -1
```

- 복잡도 : 입력의 구성에 따라 다름
  - 최선: $T_{best}(n) = 1 \ \in O(1)$
  - 최악: $T_{worst}(n) = n \ \in O(n)$
  - 평균: $T_{avg}(n) = (n+1)/2 \ \in O(n)$
    - 리스트의 모든 숫자가 골고루 한 번씩 key로 사용되는 경우를 가정

3.3 문자열 매칭

길이가 n인 입력 문자열 T와 길이가 m인 패턴 문자열 P가 있다. T에서 가장 먼저 나타나는 P의 위치를 찾아라. 패턴이 없으면 -1을 반환하라.

- 예:

	0	1	2	3	4	5	6	7	8	9
텍스트	B	R	U	T	E	F	O	R	C	E
패턴						F	O	R		

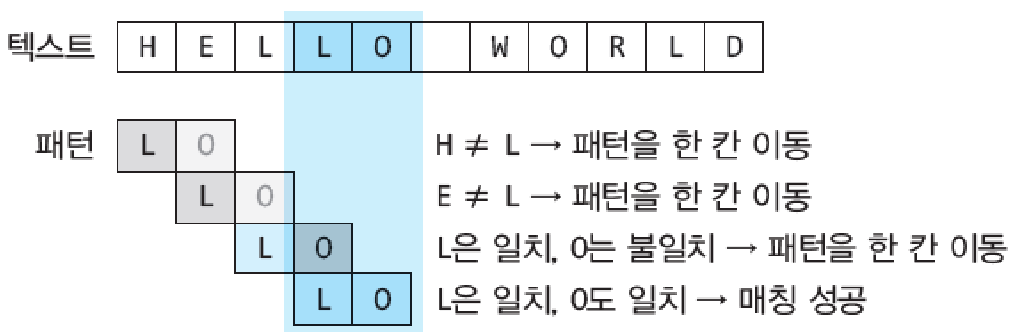
매칭 성공 → 위치(5) 반환

	0	1	2	3	4	5	6	7	8
텍스트	A	L	G	O	R	I	T	H	M
패턴	F	O	R						

매칭 실패 → -1 반환

억지 기법 문자열 매칭 전략

텍스트의 첫 번째 문자 위치에 패턴을 놓고 비교한다. 이 과정을 패턴을 오른쪽으로 한 칸씩 옮기면서 성공한 매칭이 나타날 때까지 반복한다.



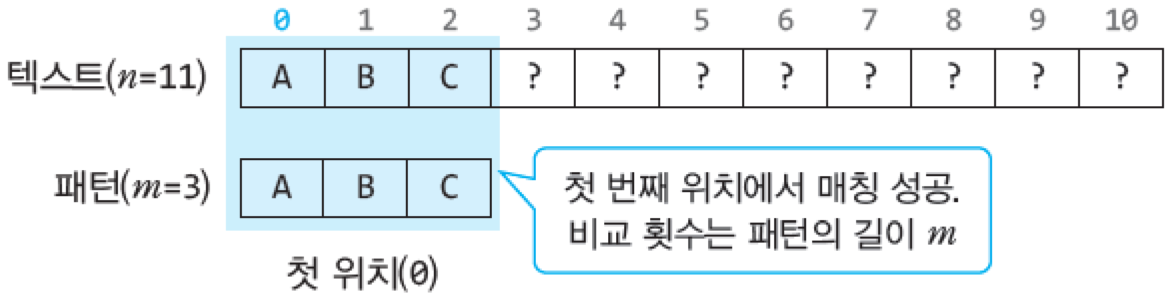
[그림 3.5] 억지 기법의 문자열 탐색 과정 예

알고리즘 3.3 문자열 매칭(억지 기법)

```
def string_matching( T, P ):
    n = len(T)
    m = len(P)
    for i in range(n-m+1) :
        j = 0
        while j < m and P[j]==T[i+j] :
            j = j + 1
        if j == m :
            return i
    return -1
```

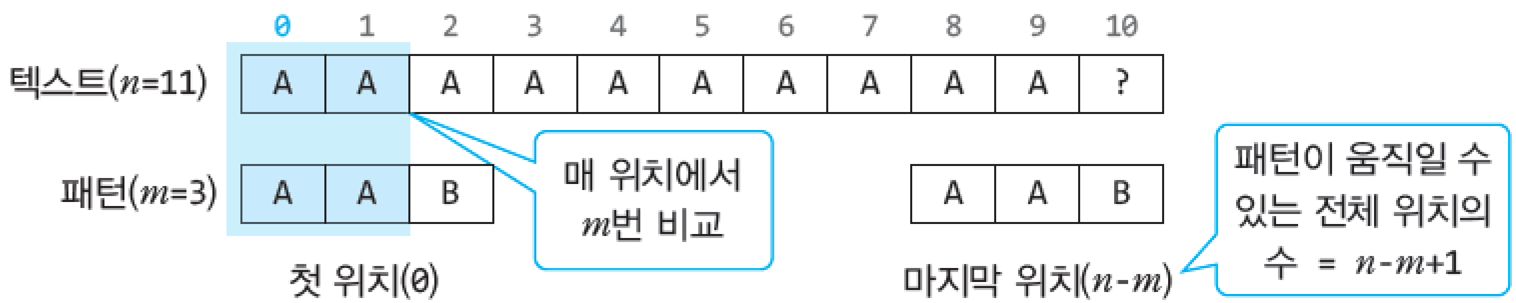
복잡도 분석

- 최선의 경우?



→ $O(m)$

- 최악의 경우?

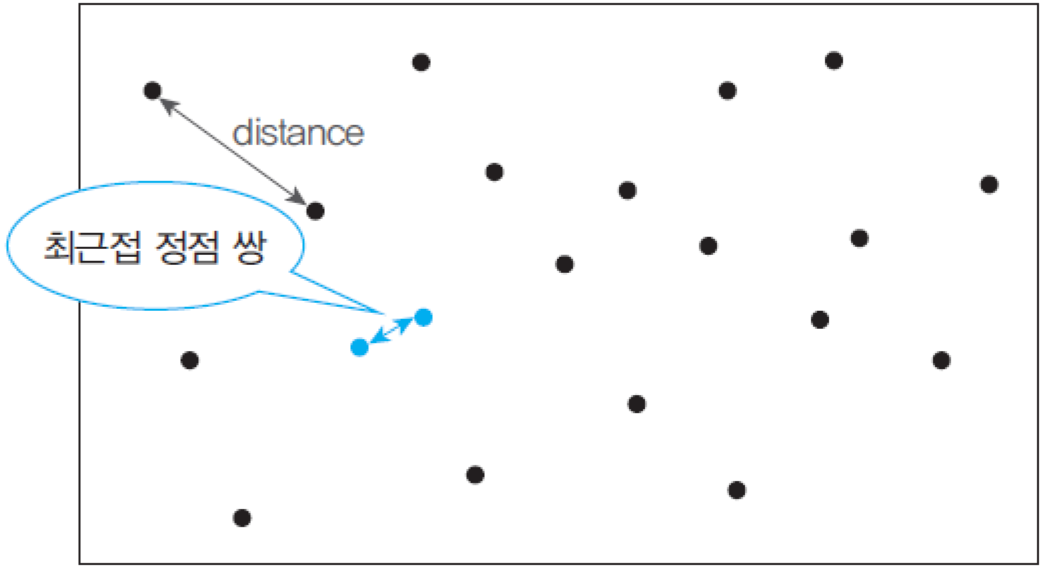


[그림 3.7] 알고리즘 3.3을 위한 최악의 입력 예

→ $O(mn)$

3.4 최근접 쌍의 거리

2차원 평면상에 n개의 점이 있다. 가장 인접한 쌍의 거리를 구하라.



[그림 3.8] 최근접 쌍의 거리 문제

- 실생활에서 점들
  - 공항의 비행기, 도로상의 자동차, 우체국의 위치, 데이터베이스 레코드, DNA 샘플 등 다양한 객체들로 대응됨

억지기법

- 거리(distance)
  - 유클리드 거리(Euclidean distance) 사용

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

    - 파이썬의 math 모듈 이용

- 억지 기법 전략

가능한 모든 점의 쌍 (p_i, p_j)에 대해 거리를 계산하고, 가장 짧은 것을 찾는다.

알고리즘과 복잡도

알고리즘 3.4 최근접 점의 쌍의 거리

```python
def closest_pair(p):
    n = len(p)
    mindist = float("inf")
    for i in range(n-1):
        for j in range(i+1, n):
            dist = distance(p[i], p[j])
            if dist < mindist:
                mindist = dist
    return mindist
```

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-i-1) \\
&= (n-1)+(n-2)+\cdots+2+1 \\
&= \frac{n(n-1)}{2} \in O(n^2)
\end{aligned}
$$

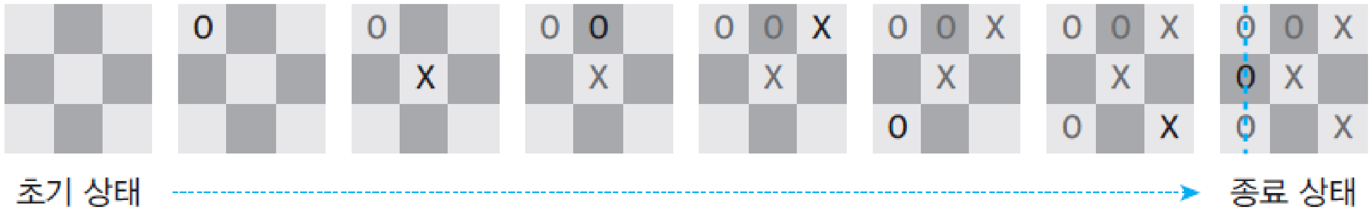
- 알고리즘 개선: 5장

3.5 완전 탐색(Exhaustive search)

- 순열이나 조합, 또는 모든 부분 집합을 찾는 과정을 포함하는 문제들이 많음
  - 순열/조합/부분집합 복습: {1, 2, 3}
    - 순열: (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)으로 총 6가지 $(n!)$
    - 조합(2개를 뽑는 조합): (1,2), (1,3), (2,3)으로 총 3가지 $({}_nC_k)$
    - 모든 부분집합: {}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}으로 총 8가지 (2^n)
- 완전 탐색(Exhaustive search)

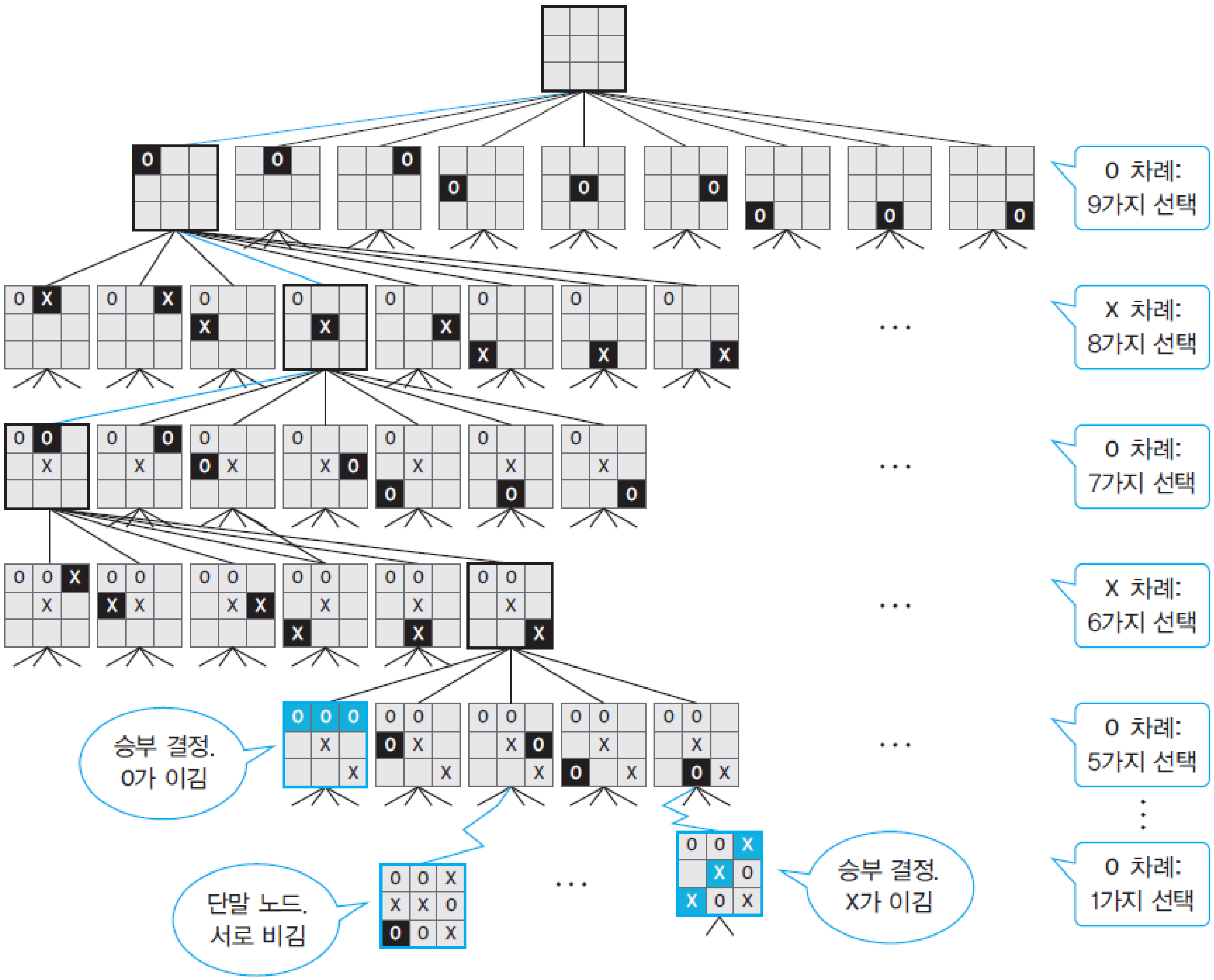
완전 탐색은 주어진 문제에 대한 상태공간트리의 모든 노드를 탐색하여 문제에 대한 해를 찾는다.

- 예) 틱택토 게임



[그림 3.9] 틱택토 게임의 예

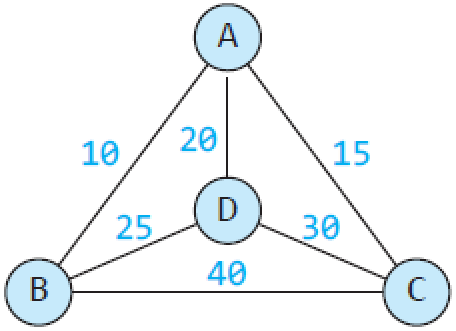
틱택토 게임의 상태공간트리



[그림 3.10] 택택토 게임의 전체 상태공간트리

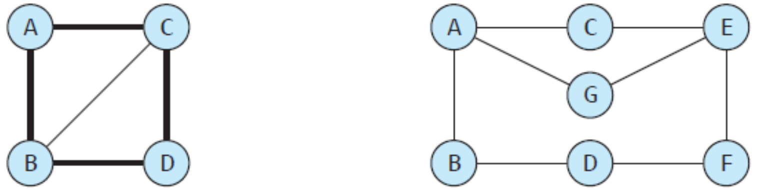
외판원 문제(Traveling Salesman Problem)

가중치 그래프 $G=(V,E)$가 주어졌다. G에서 모든 가능한 해밀토니안 사이클 중에서 경로의 합이 최소인 사이클의 경로의 합을 구하라.



경로 A,B,C,D,A : 길이=10+40+30+20=100
경로 A,D,B,C,A : 길이=20+25+40+15=100
...
경로 A,B,D,C,A : 길이=10+25+30+15=80 (TSP 경로)

[그림 3.12] 가중치 그래프에서의 TSP 경로 문제

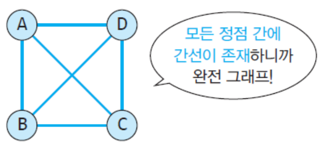


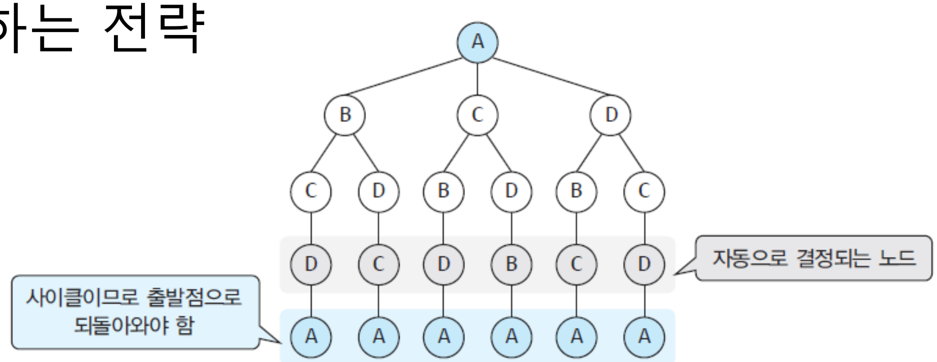
해밀토니안 사이클이 있는 그래프 해밀토니안 사이클이 없는 그래프

[그림 3.11] 해밀토니안 사이클이 있는 그래프와 없는 그래프의 예

TSP 완전탐색

- 그래프가 *N*개의 정점을 갖는 완전 그래프라면
  - 모든 해밀토니안 사이클은 $(N\text{-}1)!$개가 된다
  - 완전 탐색은 상태공간트리의 모든 단말 노드를 검사하여 길이가 최소인 것을 선택하는 전략





[그림 3.13] N=4인 완전 그래프에 대한 TSP의 상태공간트리: 3!개의 단말 노드

```
임의의 도시를 출발점으로 잡는다.
이 도시를 시작으로 하여 (N-1)!가지의 순열 객체를 생성한다.
모든 순열 객체에서 경로의 합을 구하고, 경로의 합이 최소인 순열 객체와 경로 합을 찾는다.
최소 경로의 합을 반환한다.
```

- 복잡도: $O(n!)$
  - 개선→ 백트래킹과 분기한정(9장), 근사 알고리즘(10장)

0-1 배낭 채우기 문제(Knapsack Problem)

각각 무게가 wt_i 이고 가치가 val_i 인 n개의 물건들이 있고, 이것을 배낭에 넣으려고 한다. 이때, 배낭에 넣을 수 있는 용량(최대 무게)은 W를 초과하지 않아야 하고, 물건들은 잘라서 일부분만 넣을 수는 없다. 물건들의 가치의 합이 최대가 되도록 배낭을 채우고, 이때 배낭의 최대가치를 구하라.

- 예: (10, 60), (20, 100), (30, 120)인 세 물건 A, B, C

넣는 물건	A	B	C	A, B	B, C	A, C	A, B, C
무게 합	10	20	30	30	50	40	60
가치 합	60	100	120	160	220	180	280

- 복잡도: $O(2^n)$
  - 개선→ 동적 계획법(7장)

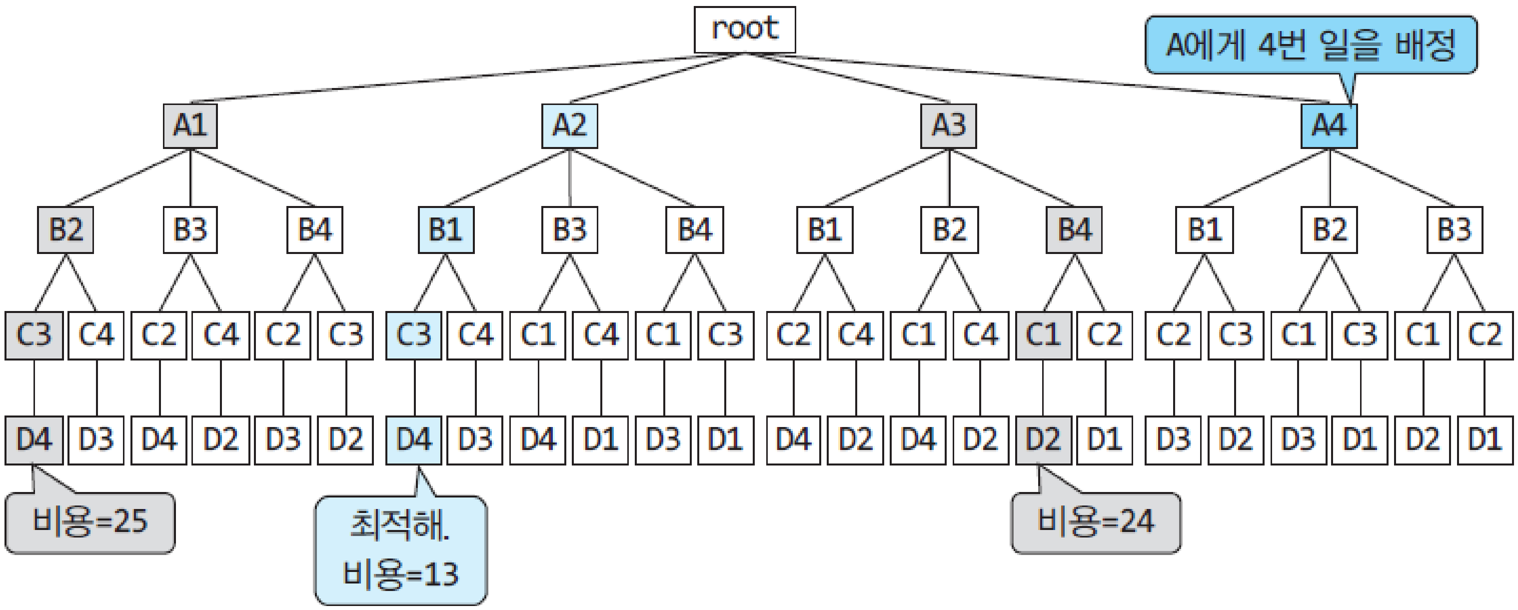
일 배정 문제(job assignment problem)

n명의 근로자와 n가지의 일이 있다. 각 근로자는 모든 일을 다 처리할 수 있지만 비용은 각각 다르게 산정되어 있다. 전체 비용을 최소화하면서 모든 근로자에게 하나씩의 일을 배정하는 경우의 전체 비용을 계산하라.

	Job1(도배)	Job2(미장)	Job3(페인트)	Job4(타일)
A(김도배)	9	2	6	8
B(이타일)	6	4	3	7
C(박타일)	5	7	1	9
D(최인트)	7	6	8	4

- A–Job1, B–Job2, C–Job3, D–Job4: 비용 = 9+4+1+4=18
- A–Job2, B–Job3, C–Job1, D–Job4: 비용 = 2+3+5+4=14
- A–Job2, B–Job1, C–Job3, D–Job4: 비용 = 2+6+1+4=13 (이 문제의 최적해)

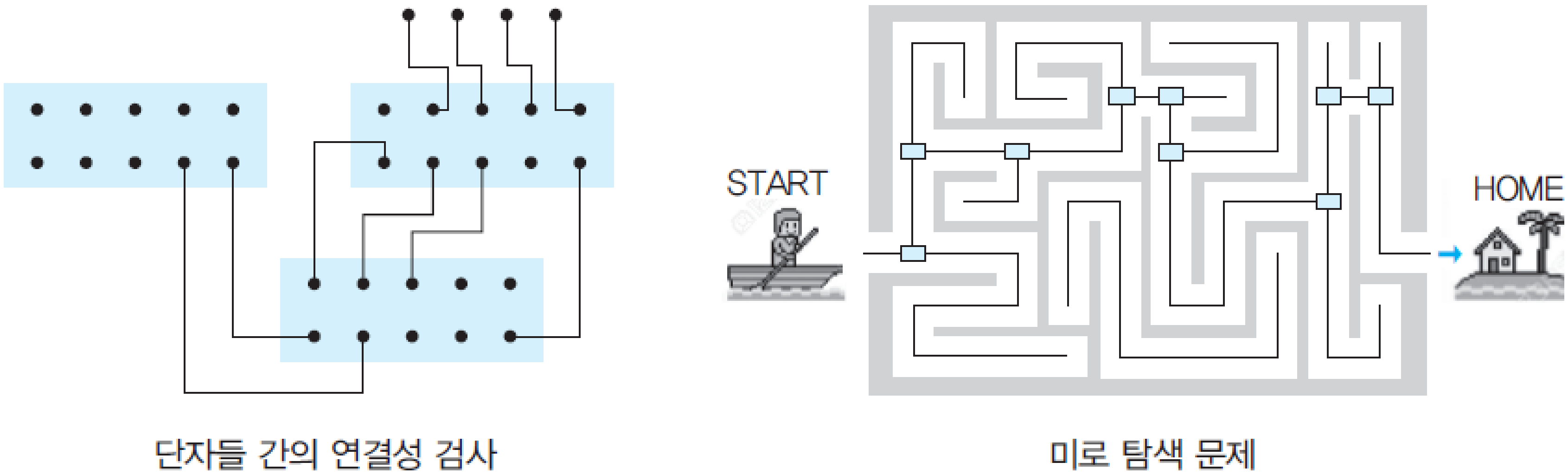
일 배정 문제 상태공간트리



[그림 3.15] N=4인 일 배정 문제의 전체 상태공간트리와 몇 개의 단말 노드의 비용

- 복잡도: $O(n!)$

3.6 그래프 탐색

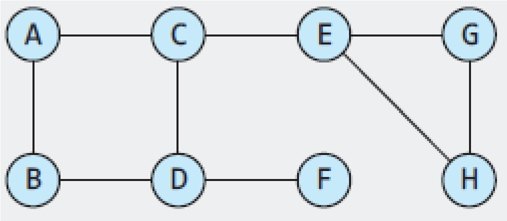


[그림 3.16] 그래프 탐색의 응용 분야

그래프 순회에 완전 탐색의 개념을 적용하면 모든 정점을 체계적으로 방문할 수 있는 두 가지 중요한 방법을 얻을 수 있다. 이것은 깊이 우선 탐색(depth first search, DFS)과 너비 우선 탐색(breadth first search, BFS)이다.

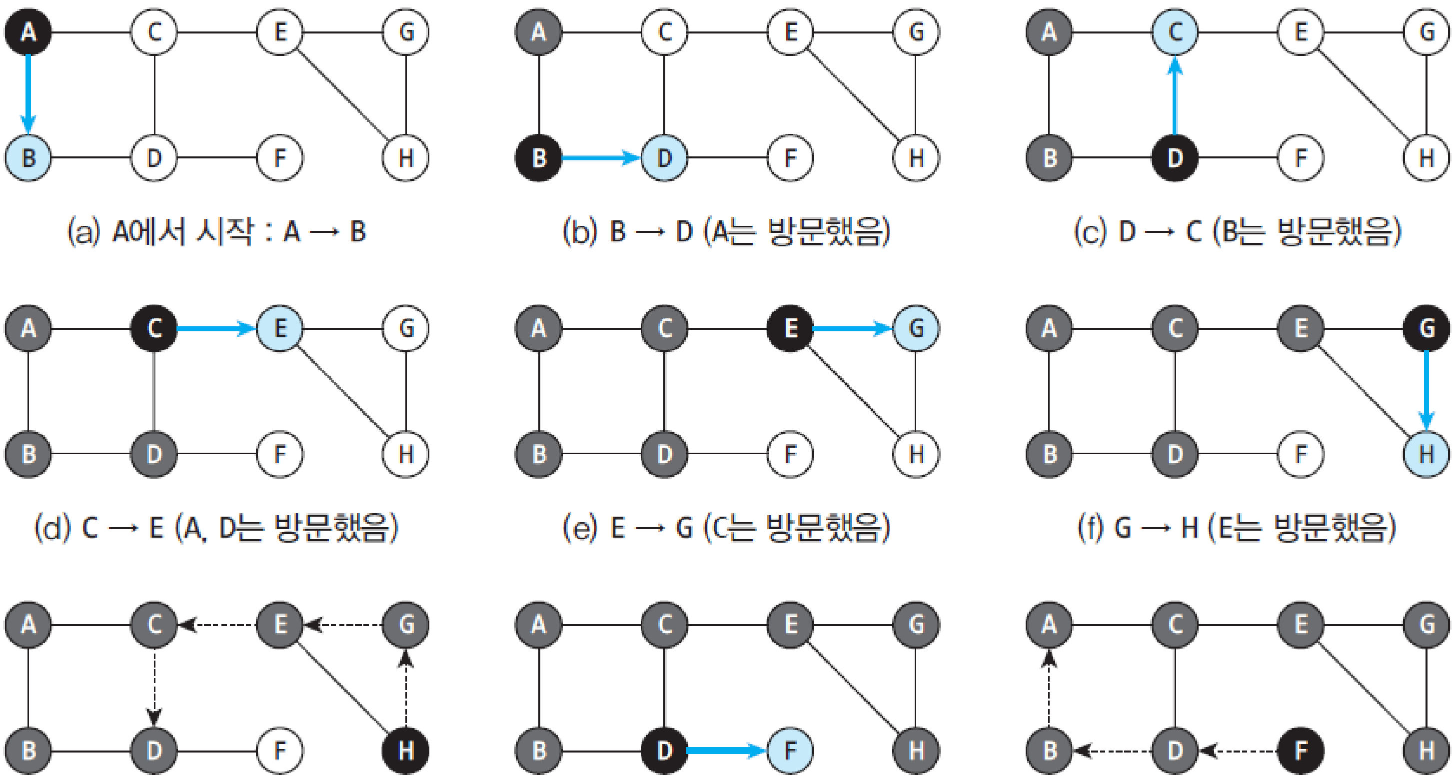
그래프의 표현: 딕셔너리와 집합 이용

- 보다 직관적으로 그래프를 표현하자.
  - 인접 리스트 → 인접 집합
  - 딕셔너리 추가 사용




```
mygraph = { "A" : {"B","C"} ),
            "B" : {"A", "D"} ),
            "C" : {"A", "D", "E"} ),
            "D" : {"B", "C", "F"} ),
            "E" : {"C", "G", "H"} ),
            "F" : {"D"} ),
            "G" : {"E", "H"} ),
            "H" : {"E", "G"} )
            }
```

깊이 우선 탐색(depth first search, DFS)



(a) A에서 시작 : A → B

(b) B → D (A는 방문했음)

(c) D → C (B는 방문했음)

(d) C → E (A, D는 방문했음)

(e) E → G (C는 방문했음)

(f) G → H (E는 방문했음)

(g) H에서는 모두 방문했음.
G, E, C, D순으로 되돌아 감.
D에서는 가지 않은 F가 있음.

(h) D → F

(i) F에서도 모두 방문했음.
D, B, A순으로 되돌아 감.
탐색 완료.
방문 순서: ABDCEGHF

[그림 3.17] 깊이 우선 탐색을 이용한 정점 방문 과정

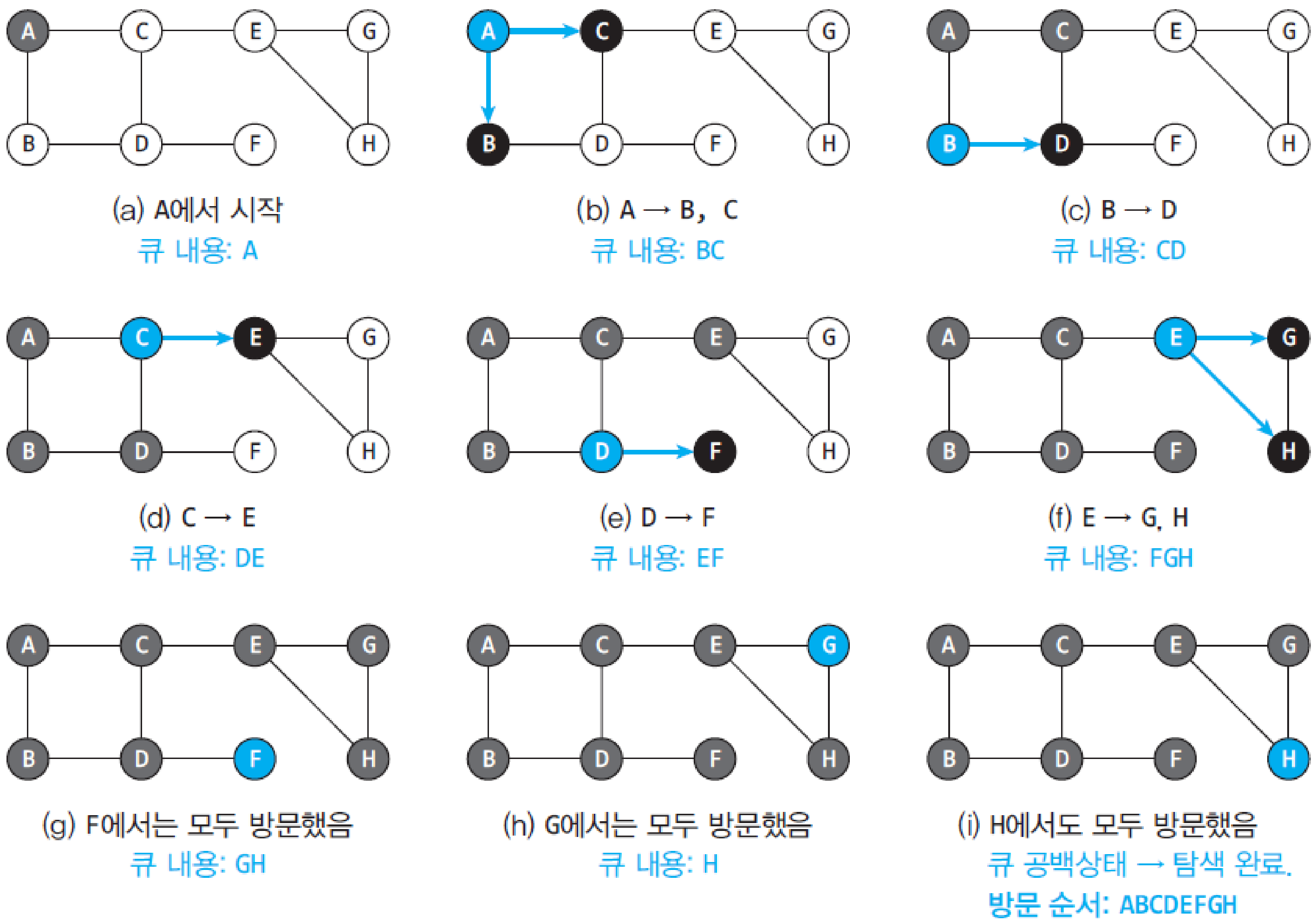
DFS 알고리즘

알고리즘 3.6 깊이 우선 탐색

```
def dfs(graph, start, visited ):
    if start not in visited :
        visited.add(start)
        print(start, end=' ')
        nbr = graph[start] - visited
        for v in nbr:
            dfs(graph, v, visited)
```

- 정점의 수 n, 간선의 수 e인 경우
  - 인접 리스트 표현: $O(n+e)$ → 희소 그래프에서 유리
  - 인접 행렬 표현: $O(n^2)$

너비 우선 탐색(breadth first search: BFS)



[그림 3.18] 너비 우선 탐색을 이용한 정점 방문 과정

BFS 알고리즘

- 큐 사용
  - 파이썬의 queue모듈

알고리즘 3.7 너비 우선 탐색

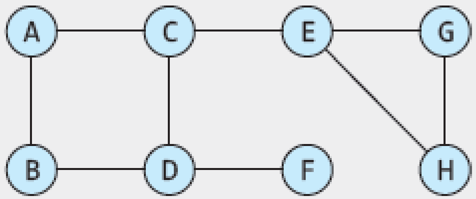
```
import queue
def bfs(graph, start):
    visited = { start }
    que = queue.Queue()
    que.put(start)
    while not que.empty():
        v = que.get()
        print(v, end=' ')
        nbr = graph[v] - visited
        for u in nbr:
            visited.add(u)
            que.put(u)
```

- 복잡도
  - 인접 리스트 표현: $O(n+e)$
  - 인접 행렬 표현: $O(n^2)$

테스트

알고리즘 테스트 깊이 우선 탐색 테스트

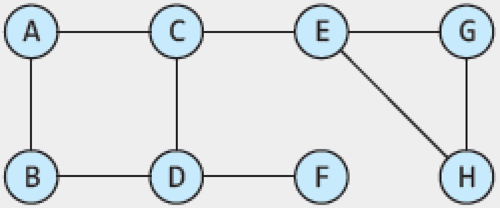
```
mygraph = { "A" : {"B","C"} ), ... }


print('DFS : ', end='')
dfs(mygraph, "A", set() )          # visited에 공집합 전달
print()
```




```
C:\WINDOWS\system32\cmd.exe
BFS : A B C D E F G H
```

알고리즘 테스트 너비 우선 탐색 테스트

```
mygraph = { "A" : {"B","C"} ), ... }
print('BFS : ', end='')
bfs(mygraph, "A")
print()
```




```
C:\WINDOWS\system32\cmd.exe
BFS : A B C D E F G H
```

실습 과제

감사합니다!