



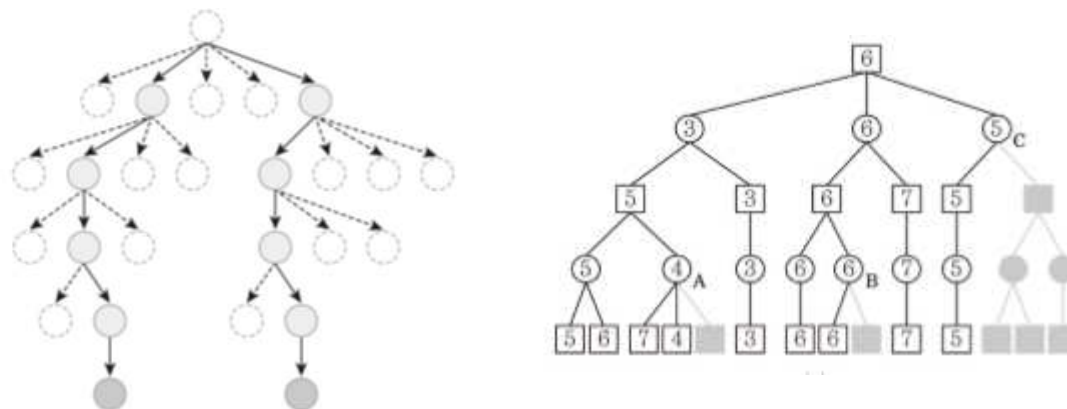
제2장 탐색

이번 장에서 다루는 내용

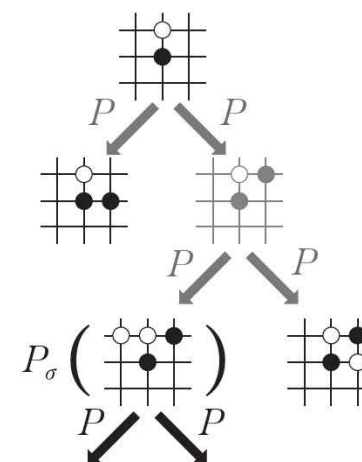
- 탐색의 개념을 소개한다.
- 상태, 상태 공간, 연산자의 개념을 소개한다.

인공 지능의 연구 분야 - 요소 기술

- 탐색(search)
 - 문제의 답이 될 수 있는 것(상태)들의 집합을 공간(space)으로 간주하고, 문제에 대한 최적의 해(경로 또는 상태)를 찾기 위해 공간을 체계적으로 찾아 보는 것
 - 무정보 탐색
 - 너비우선 탐색(breadth-first search), 깊이우선 탐색(depth-first search), 반복적 깊이증가 탐색
 - 휴리스틱 탐색
 - 언덕오르기 탐색, 최선 우선탐색, 빔탐색(Beam search), A* 알고리즘
 - 게임 트리 탐색
 - mini-max 알고리즘, α - β 가지치기(pruning), 몬테카를로 트리 탐색



- 알파고는 딥러닝과 탐색 기법을 통하여 다음 수를 읽었다.



d Update(갱신)

상태, 상태공간, 연산자

- 탐색(search)이란 상태공간에서 시작상태에서 목표상태까지의 경로를 찾는 것
- 상태공간(state space): 상태들이 모여 있는 공간
- 연산자: 다음 상태를 생성하는 것
- 초기상태
- 목표상태

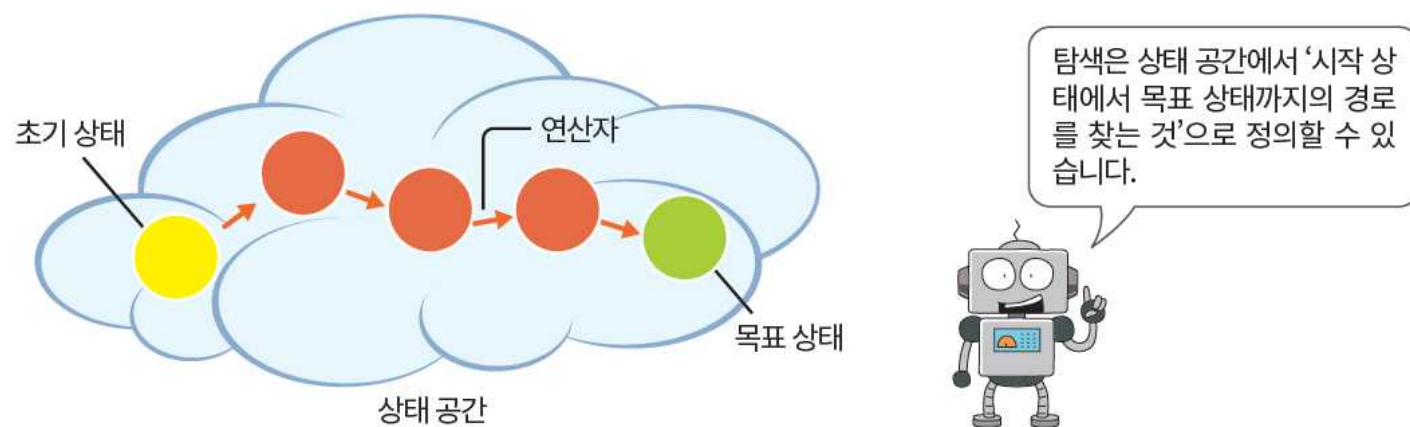
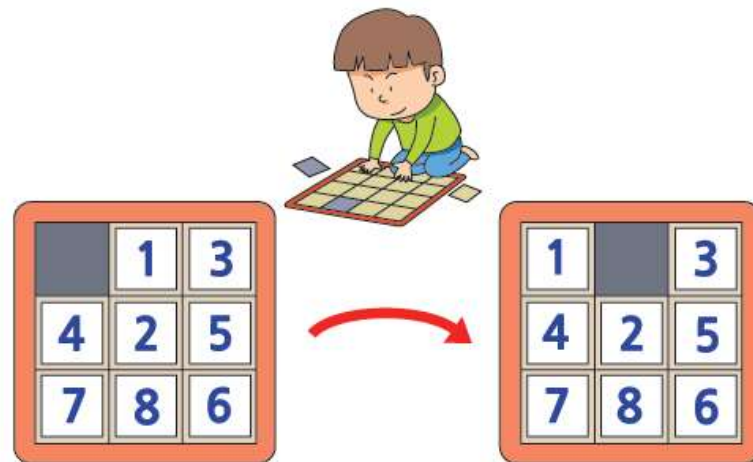
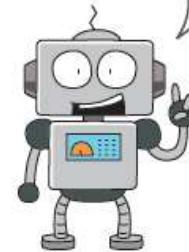


그림 2-2 상태, 상태 공간, 연산자

8-puzzle



8-퍼즐은 슬라이딩 퍼즐의 일종으로, 타일을 움직여서 순서대로 맞추는 퍼즐 놀이입니다.



8-puzzle

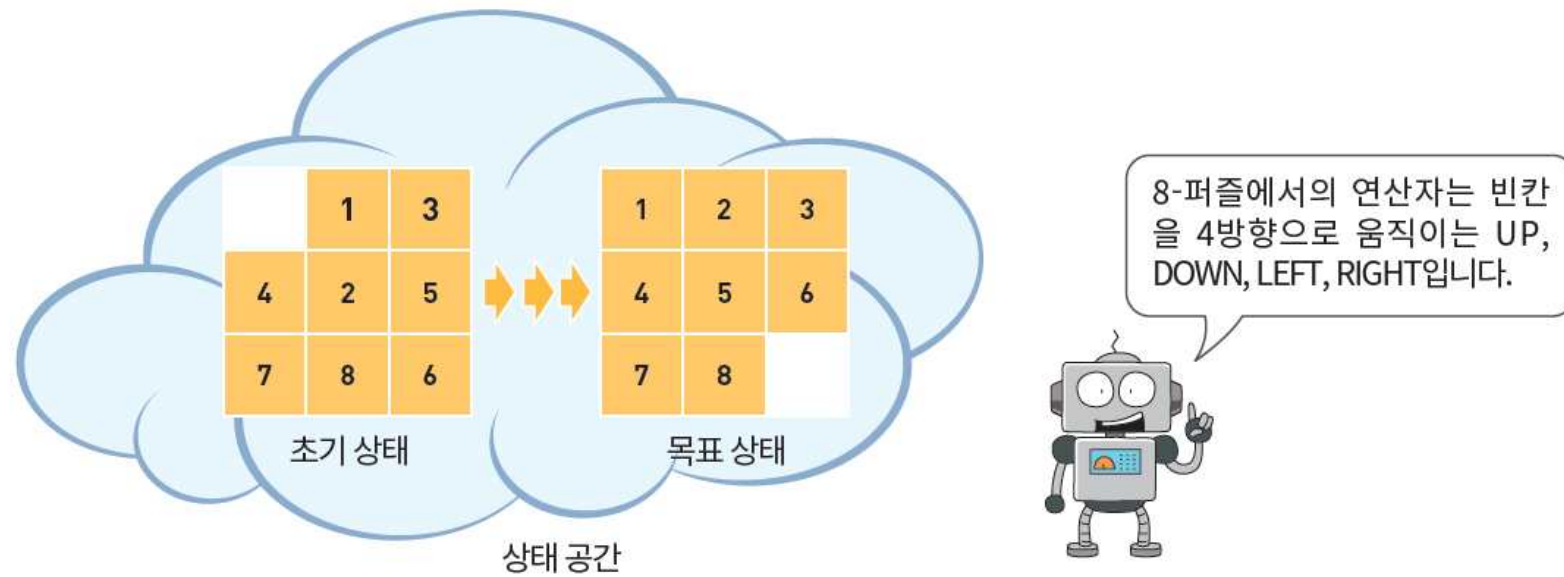
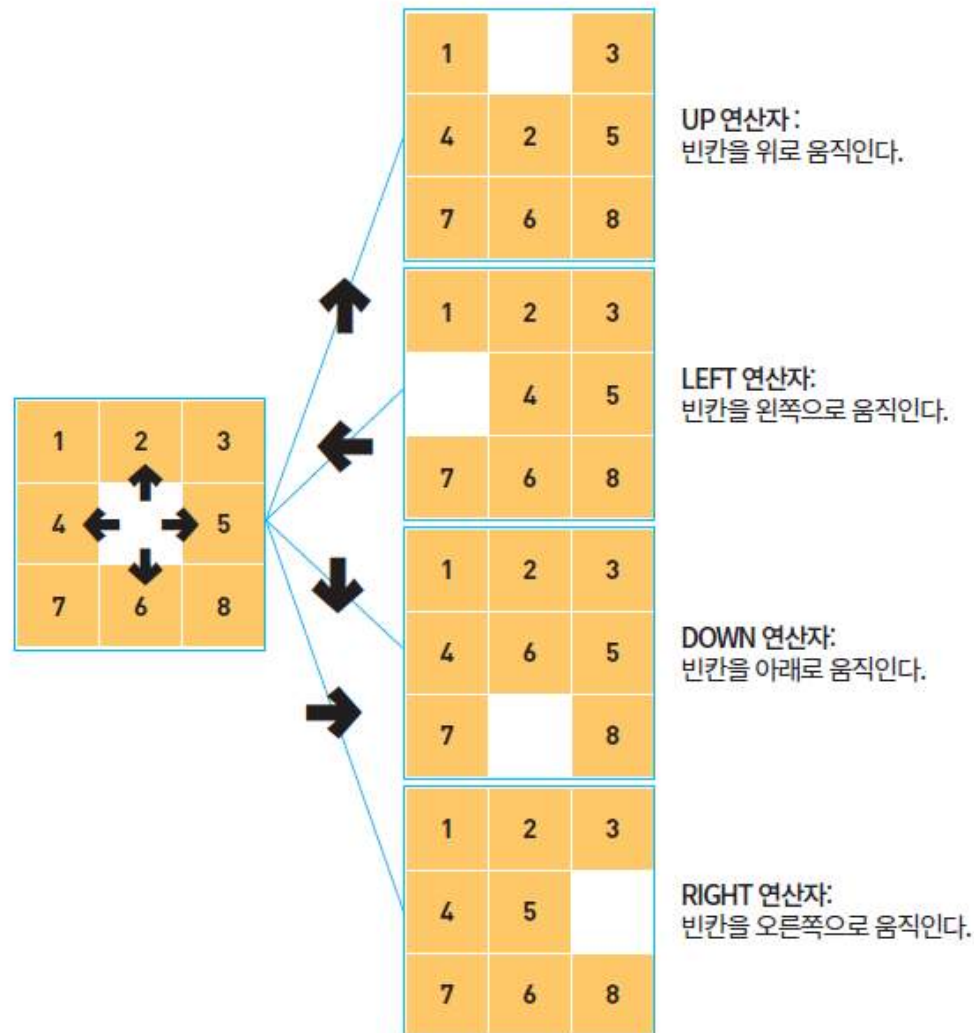


그림 2-3 8-퍼즐에서의 상태 공간

8-puzzle에서의 연산자



편의상 타일을 움직이는 것이 아니라 빈칸을 움직인다고 생각합니다. 그래야 연산자를 4개로 제한할 수 있습니다.



그림 2-4 상하좌우 연산자

8-puzzle 에서의 상태 공간

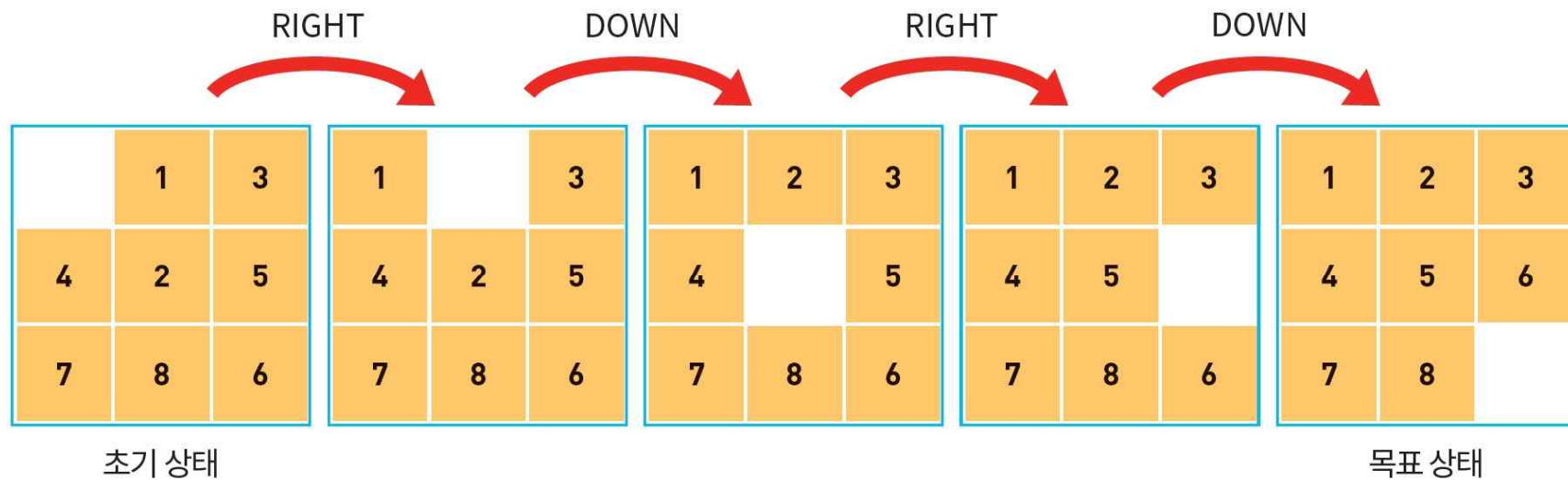
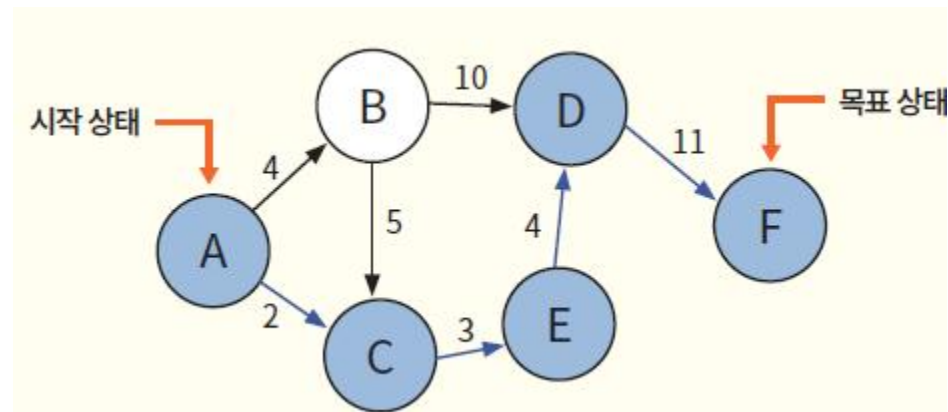


그림 2-5 연산자를 사용하여 이동하는 8-퍼즐 예제

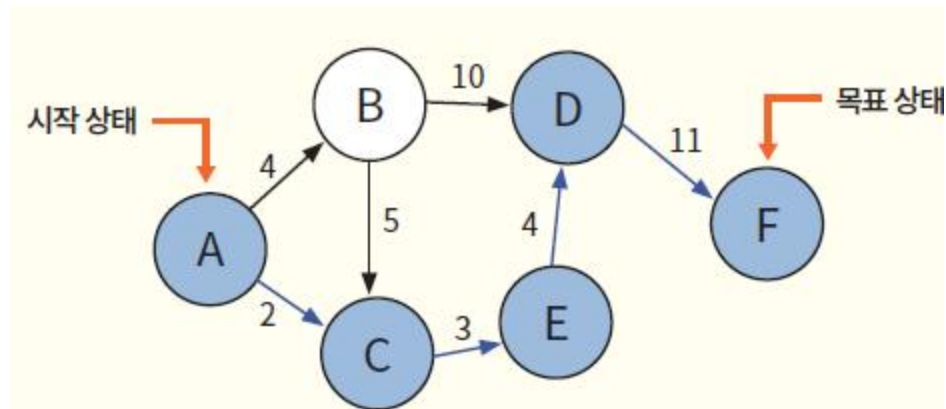
Lab: 경로 찾기 문제

- 탐색의 대표적인 문제 중의 하나가 경로를 찾는 문제이다. 예를 들어서 다음과 같이 도시들이 연결되어 있다고 하자.
- 상태?
- 연산자?



Lab: 경로 찾기 문제

- 이 문제에서 초기 상태, 목표 상태, 연산자 등을 생각해보자.
- 초기 상태는 에이전트가 **A**에 있는 것이다.
- 목표 상태는 에이전트가 **F**에 있는 것이다.
- 연산자는 가능한 경로 중에서 하나의 경로를 선택하는 것이다.



Lab: 하노이 탑

- 상태?
- 연산자?

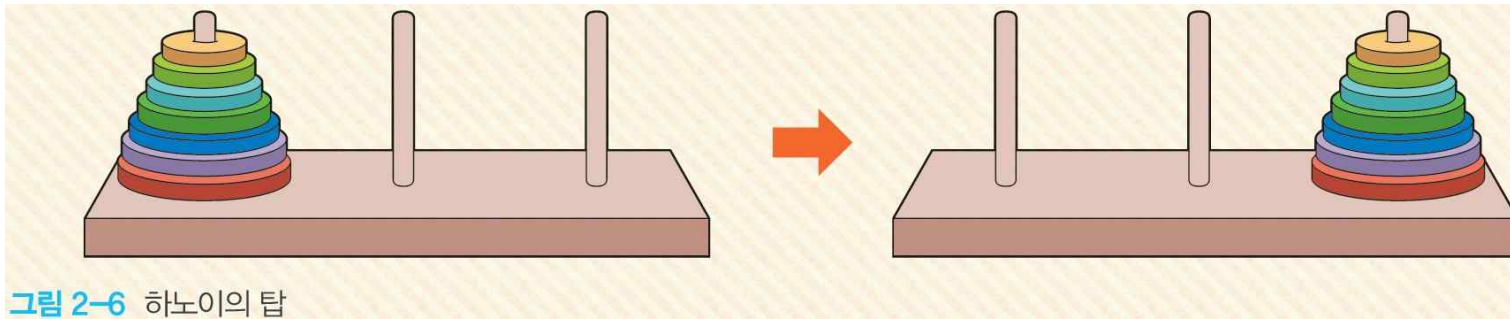


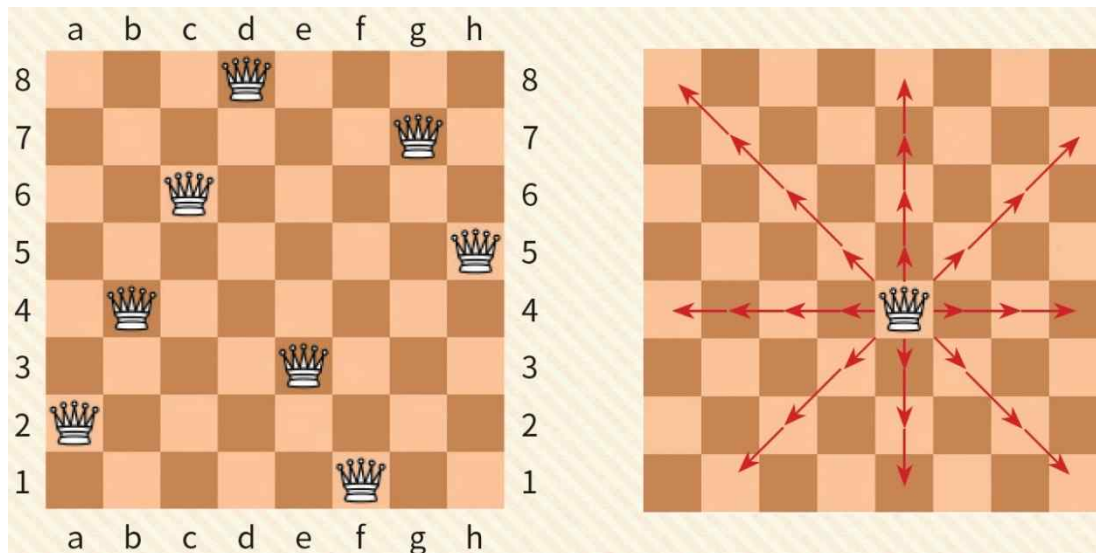
그림 2-6 하노이의 탑

Lab: 하노이 탑

- 원반이 세개인 경우
- 상태공간 $A = \{ (a_1, a_2, a_3) \mid a_i \in \{A, B, C\} \}$
 - 각 원반이 어떤 기둥에 있는지를 표시
- 초기상태 $I = (A, A, A)$
- 목표 상태 $G = (C, C, C)$
- 연산자 $O = \{ \text{move}_{\text{which, where}} \mid \text{which} \in \{1, 2, 3\}, \text{where} \in \{A, B, C\} \}$

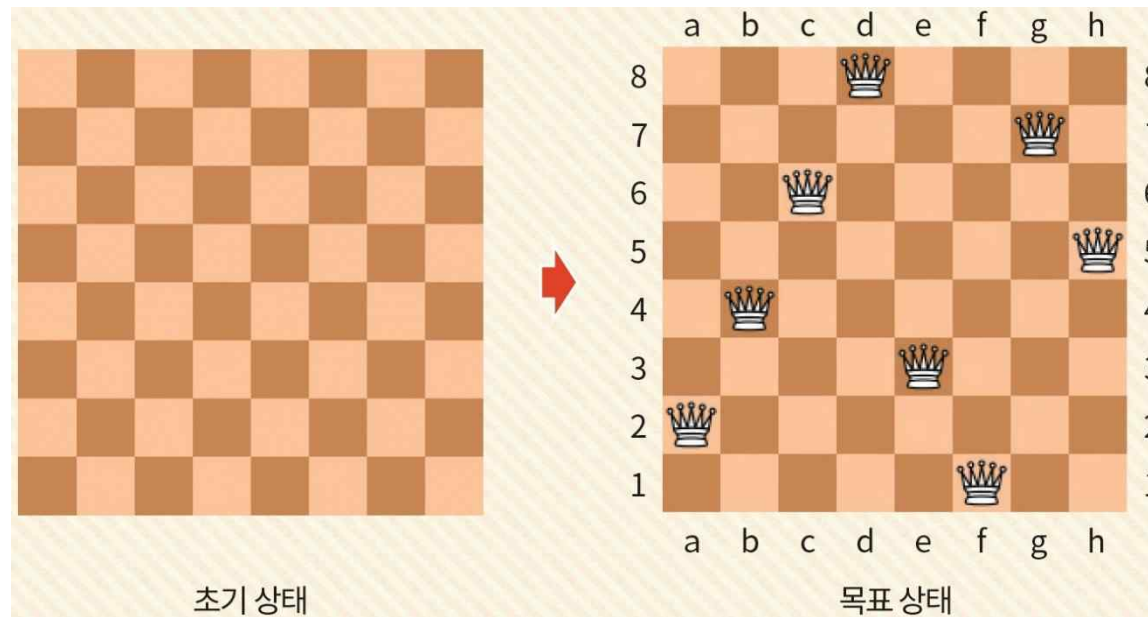
Lab:N-queen

- 8-queen 문제는 8×8 체스판에 두 개의 퀸이 서로를 위협하지 않도록 8개의 퀸을 배치하는 문제이다.
- 상태?
- 연산자?



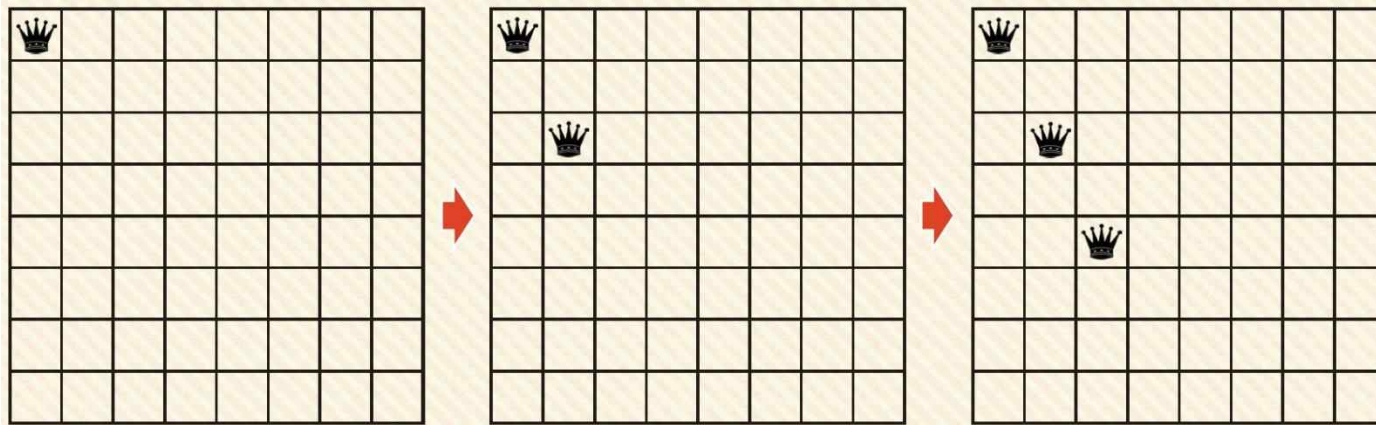
Lab:N-queen

- 초기 상태와 목표 상태



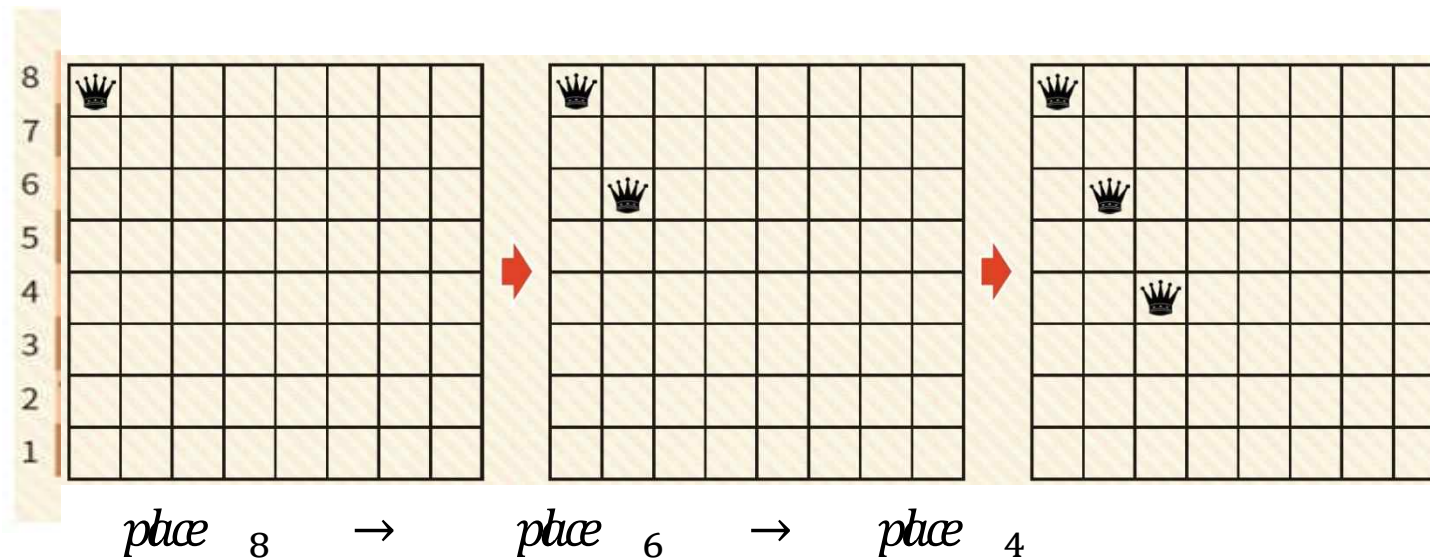
Lab:N-queen

- 각각의 퀸을 정해진 열에서만 움직이게 한다.



Lab:N-queen

- 연산자 집합 $O = \{ place_i \mid 1 \leq i \leq 8 \}$
- $place_i$ 연산자는 새로운 퀸을 i 번째 행에 배치한다.



탐색 트리

- 상태 = 노드(node)
- 초기 상태 = 루트 노드
- 연산자 = 간선(edge)

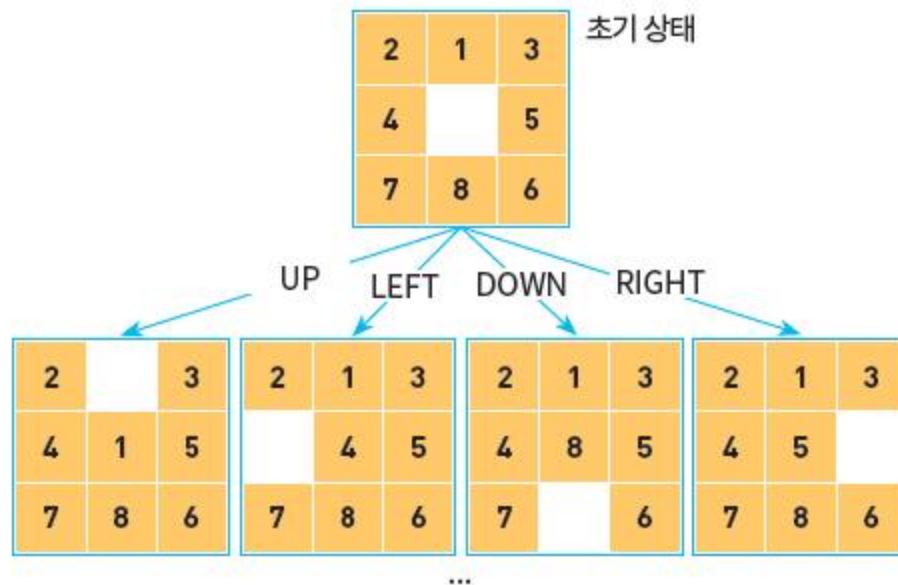


그림 2-8 탐색 트리



탐색 트리는 초기 상태에서 연산자를 적용하면 만들어지는 트리입니다.

탐색 트리

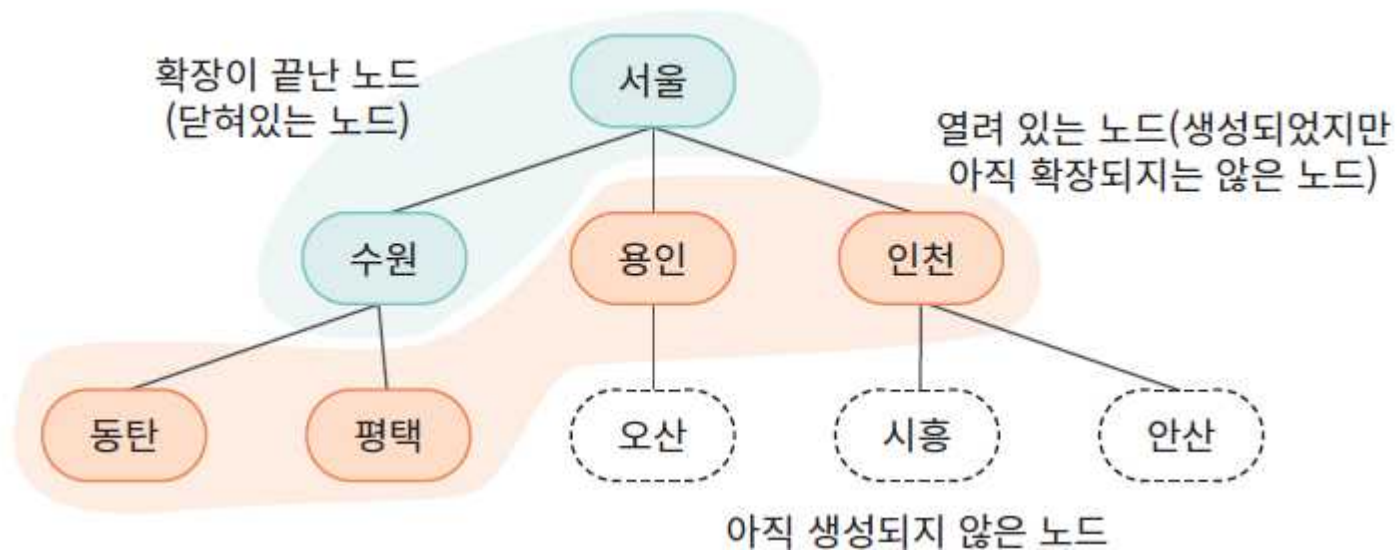
- 연산자를 적용하기 전까지는 탐색 트리는 미리 만들어져 있지 않음!



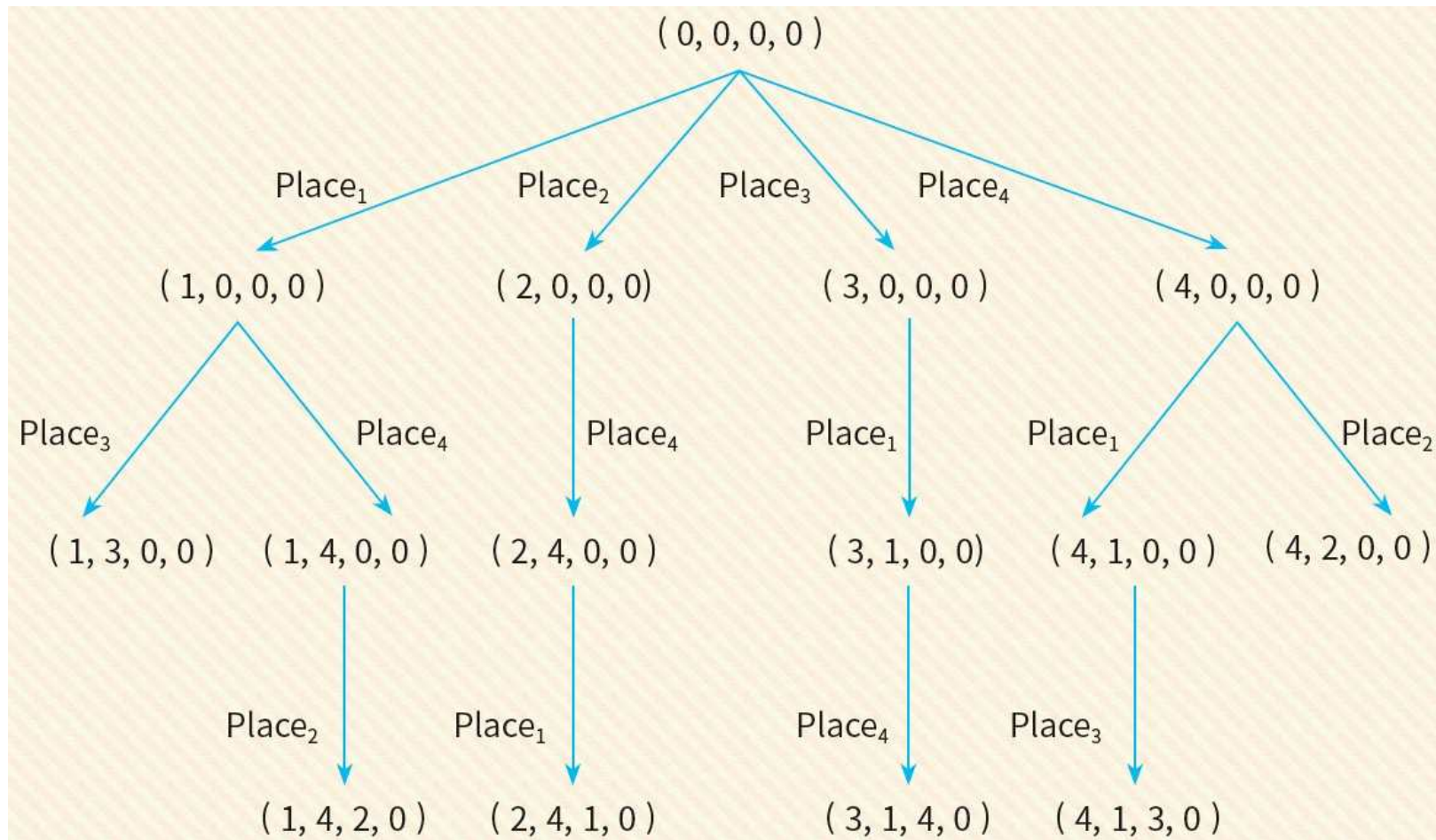
그림 2-9 탐색 트리의 노드는 동적으로 생성된다.

탐색 트리의 예

- 다음 그림은 출발 도시에서 목표 도시까지의 경로를 탐색하는 문제에서 노드들을 분류하였다.



Lab: 4-queen 문제 탐색 트리의 일부

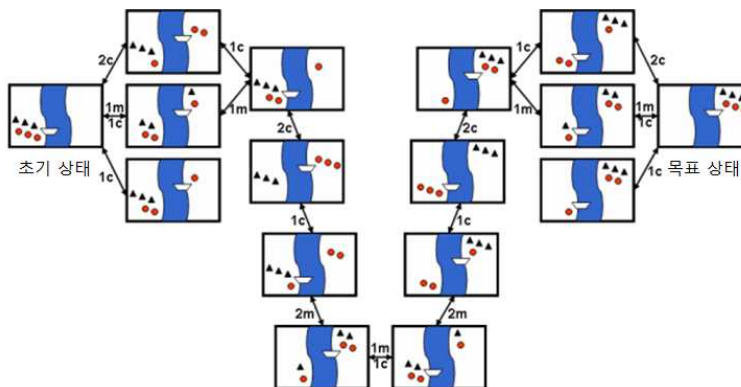


- 선교사와 식인종

- 세명의 선교사와 세명의 식인종이 강가에 왔다. 강가에는 한 사람 혹은 두 사람이 탈 수 있는 배가 한 척 있고 빈배로는 움직일 수 없다. 어떻게 하면 강 어느 편에도 식인종의 수가 선교사의 수보다 많지 않도록 하면서 강을 건널 수 있을까?
- 이 문제를 위한 상태 표현을 명시하고, 시작 상태와 목표 상태를 나타내어라.
 - 상태표현 그래프 전체를 그리고, 노드들을 나타내어라 (이 그림에는 ‘정당한’ 상태들, 즉 강 양편에 모두 식인종의 수가 선교사의 수보다 많지 않은 상태들만 포함하면 된다.
 - (3, 3, L, 0, 0) // (왼편선교사, 왼편식인종, 배위치, 오선, 오식)
 - (0, 0, R, 3, 3)

상태 공간과 탐색

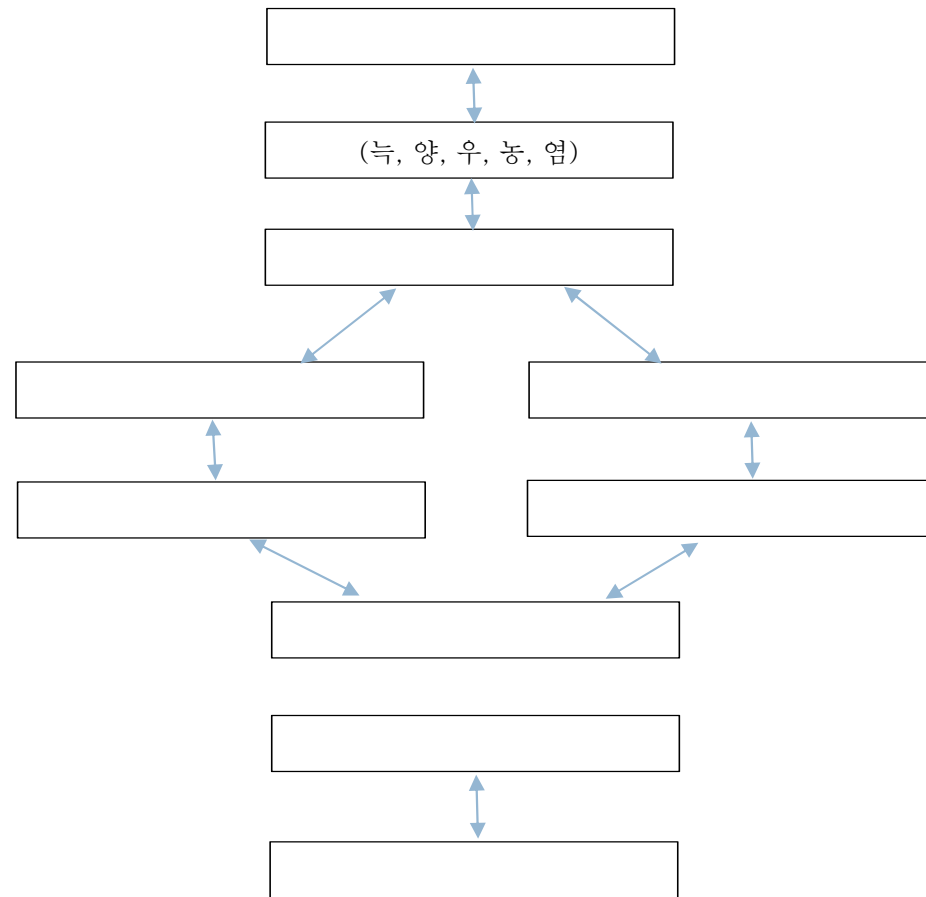
- 상태 공간 그래프(state space graph)
 - 상태공간에서 각 행동에 따른 상태의 변화를 나타낸 그래프
 - 노드 : 상태
 - 링크 : 행동
 - 선교사-식인종 문제



▲ m: 선교사
● c: 식인종

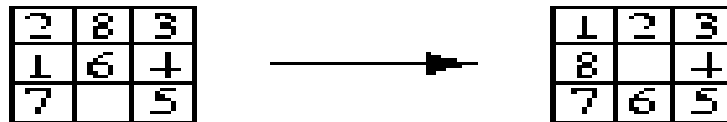
- 해(solution)
초기상태에서 목표 상태로의 경로(path)
- 일반적인 문제에서는 상태공간이 매우 큼
 - 미리 상태 공간 그래프를 만들기 어려움
 - 탐색과정에서 그래프 생성

- 다음 문제에 대한 상태공간 그래프를 작성하시오. 농부가 늑대 한 마리, 염소 한 마리, 양배추 한 꾸러미와 함께 강을 건너 집에 가야한다. 강을 건널 때는 농부와 위 세 가지 중 하나만 태울 수 있다. 농부가 없는 상태에서 늑대와 염소가 함께 있으면 늑대가 염소를 잡아 먹어버릴 수 있고, 염소와 양배추가 함께 있으면 염소가 양배추를 먹어버릴 것이다. 어떤 피해도 없이 모두가 강을 건너야한다. 상태표시는 (농, 늑, 염, 양, 좌), (우, 농, 늑, 염, 양)과 같이 하며 배의 위치를 기준으로 강의 왼쪽과 오른쪽에 있는 요소를 나누어 기록한다. 농(농부), 늑(늑대), 염(염소), 양(양배추), 좌(배가 강의 좌측), 우(배가 강의 우측).



상태공간의 설정

- 실제 문제들의 대부분은 탐색공간이 너무 큼
 - 명시적인 그래프로 표현할 수 없음
- 블록 쌓기, 타일 맞추기(4*4), ...
- 8 퍼즐에서의 시작과 목표 배치 :



- 시작 상태와 가능한 이동의 집합
 - 시작 상태에서 도달할 수 있는 암시적 상태 그래프
- 8 퍼즐의 상태공간 그래프의 노드 수
 - $9! = 362,880$

암시적인 상태공간 그래프의 구성요소

- 암시적인 상태공간 그래프
너무 방대해서 전체를 명시적으로 나타낼 수 없는 그래프의 경우, 탐색 과정은 상태 공간 중 목표까지의 경로를 찾는 데 필요한 부분만 명시적으로 만들면 된다
- 세 가지 기본 요소
 1. 시작 노드(Start node)의 표현: 초기상태 자료구조
 2. 연산자(Operators): 하나의 상태 표현을 어떤 행동에 대한 결과 상태 표현으로 바꾸어주는 함수
 3. 목표 조건(goal condition)
- 크게 두 종류의 탐색 방법
 1. 무정보 탐색(Uninformed Search):
Breadth-First Search(BFS), Depth-First Search(DFS)
Iterative Deepening Search(IDS)
 2. 휴리스틱 탐색(Heuristic Search):
Best-First Search, A*

4 기본적인 탐색 기법



그림 2-10 탐색의 종류

맹목적인 탐색 vs 경험적인 탐색

- 맹목적인 탐색 방법(**blind search method**)은 목표 노드에 대한 정보를 이용하지 않고 기계적인 순서로 노드를 확장하는 방법으로 매우 소모적인 탐색을 하게 된다.
- 경험적 탐색 방법(**heuristic search method**)은 목표 노드에 대한 경험적인 정보를 사용하는 방법이다. 따라서 효율적인 탐색이 가능하다. 이런 경험적인 정보를 휴리스틱(**heuristic**)이라고 한다.

탐색 성능 측정

- 완결성(**completeness**): 문제에 해답이 있다면, 반드시 해답을 찾을 수 있는지 여부
- 최적성(**optimality**): 가장 비용이 낮은 해답을 찾을 수 있는지 여부
- 시간 복잡도(**time complexity**): 해답을 찾는데 걸리는 시간
- 공간 복잡도(**space complexity**): 탐색을 수행하는 데 필요한 메모리의 양
 - b : 탐색 트리의 최대 분기 계수
 - d : 목표 노드의 깊이
 - m : 트리의 최대 깊이

깊이 우선 탐색(DFS)

- 깊이 우선 탐색(**depth-first search**)은 탐색 트리 상에서, 해가 존재할 가능성이 존재하는 한, 앞으로 계속 전진하여 탐색하는 방법이다.

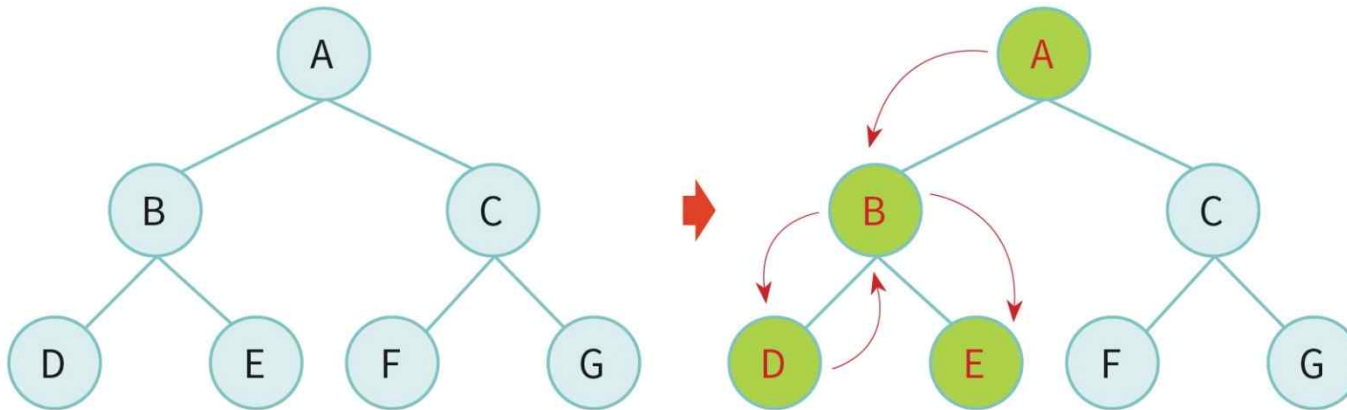


그림 2-11 깊이 우선 탐색

깊이 우선 탐색(8-puzzle)

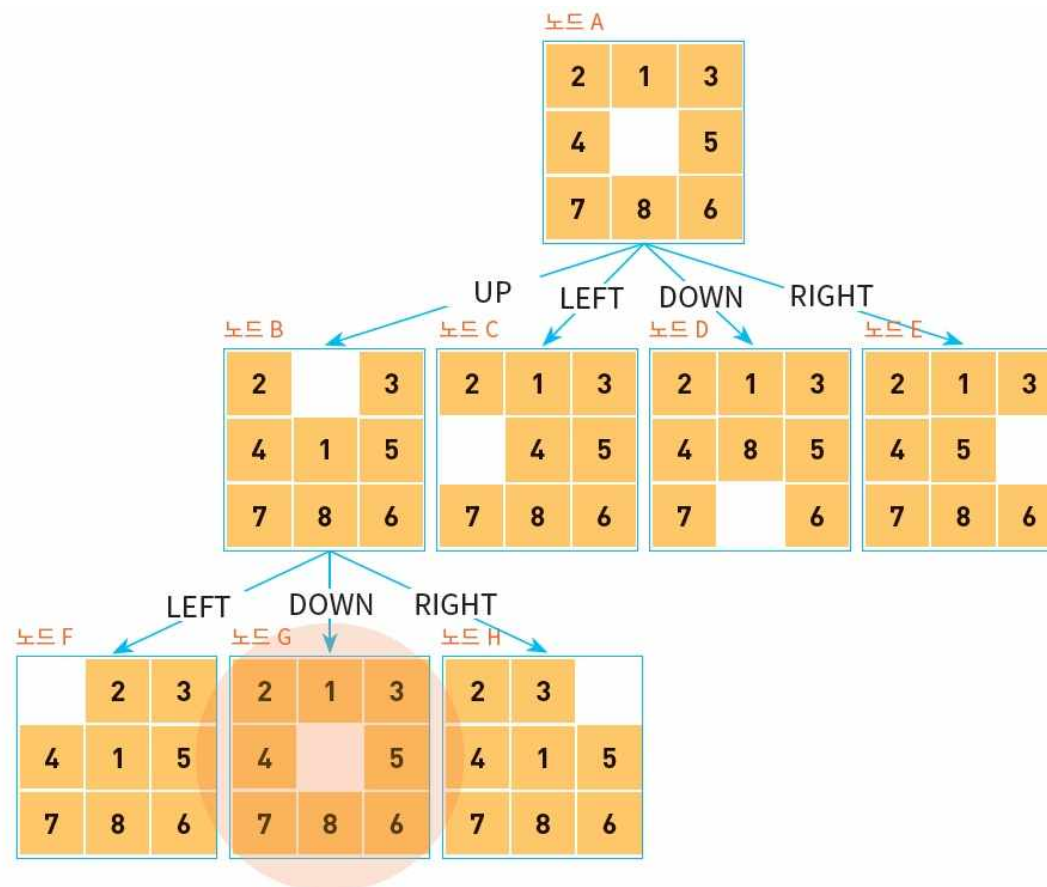
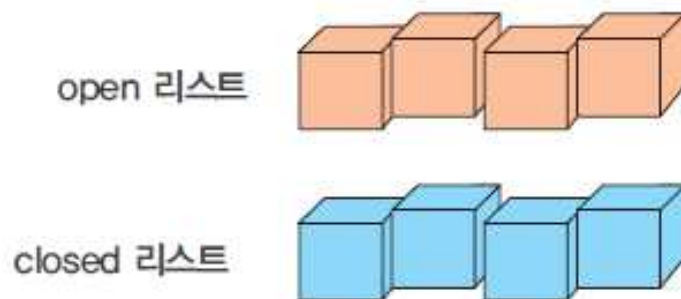


그림 2-12 8-퍼즐에서의 깊이 우선 탐색

OPEN 리스트와 CLOSED 리스트

- 탐색에서는 중복된 상태를 막기 위하여 다음과 같은 **2개의 리스트**를 사용한다.
 - **OPEN** 리스트: 확장은 되었으나 아직 탐색(자식을 만들지)하지 않은 상태들이 들어 있는 리스트
 - **CLOSED** 리스트: 탐색이 끝난(자식을 만든) 상태들이 들어 있는 리스트

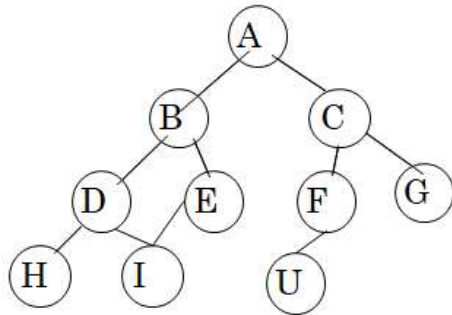


open 리스트는 frontier라고도 해요. closed 리스트는 reached 리스트라고도 합니다.

DFS 알고리즘

```
function DFS(root)
  open ← [root]
  closed ← []
  while open ≠ [] do
    X ← open 리스트의 첫 번째 요소
    if X == goal then return SUCCESS
    else
      X의 자식 노드를 생성한다.
      X를 closed 리스트에 추가한다.
      X의 자식 노드가 이미 open이나 closed에 있다면 버린다.
      남은 자식 노드들은 open의 처음에 추가한다. (스택처럼 사용)
  return FAIL
```

DFS 예: 목표가 U인 경우



1. open = [A]; closed = []
2. open = [B,C]; closed = [A]
3. open = [D,E,C]; closed = [B,A]
4. open = [H,I,E,C]; closed = [D,B,A]
5. open = [I,E,C]; closed = [H,D,B,A]
6. open = [E,C]; closed = [I,H,D,B,A]
7. open = [C]; closed = [E,I,H,D,B,A] (I는 이미 close 리스트에 있으니 추가되지 않는다.)
8. open = [F,G]; closed = [C,E,I,H,D,B,A]
9. open = [U,G]; closed = [F,C,E,I,H,D,B,A]
12. 목표 노드 U 발견!

깊이 우선 탐색의 분석

- 완결성: 무한 상태 공간에서는 깊이 우선 탐색은 무한 경로를 따라 끝없이 나갈 수 있다. 따라서 무한 상태 공간에서는 완결적이지 않다.
- 시간 복잡도: $O(b^m)$ 이다. 여기서 b 는 분기 계수이다. 만약 m (트리의 최대 깊이)이 d (정답의 깊이)보다 아주 크다면 시간이 많이 걸린다. 그렇지 않은 경우에는 너비 우선 탐색보다도 빠를 수 있다.

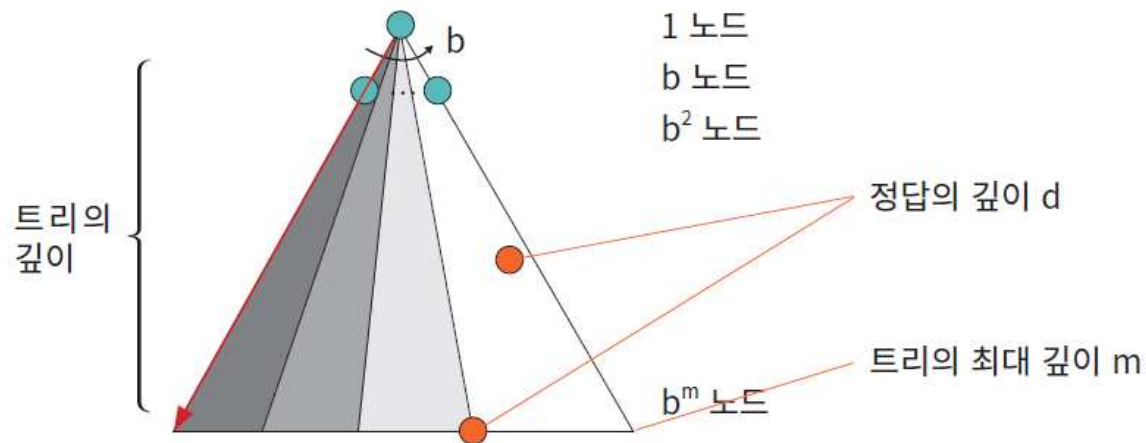


그림 2-15 DFS 시간 복잡도

깊이 우선 탐색의 분석

- 공간 복잡도: $O(bm)$, 즉, 선형 복잡도만을 가진다. 탐색 트리에서 한 줄 단위로 탐색하므로 모든 노드를 저장하고 있을 필요는 없다. 이점은 너비 우선 탐색에 비하여 큰 장점이다. 만약 b 가 10이고, 노드당 1KB가 필요하다고 하자, 만약 $m=10$ 이라고 하면 100KB 바이트만 필요하다. 만약 너비 우선 탐색이라면 10TB가 필요하다.
- 최적성: 가장 경로가 짧은 최적의 해답은 발견할 수 없다. 가장 왼쪽 (leftmost)에 있는 해답만을 발견한다.

너비 우선 탐색(BFS)

- 루트 노드의 모든 자식 노드들을 탐색한 후에 해가 발견되지 않으면 한 레벨 내려가서 동일한 방법으로 탐색을 계속하는 방법이다.

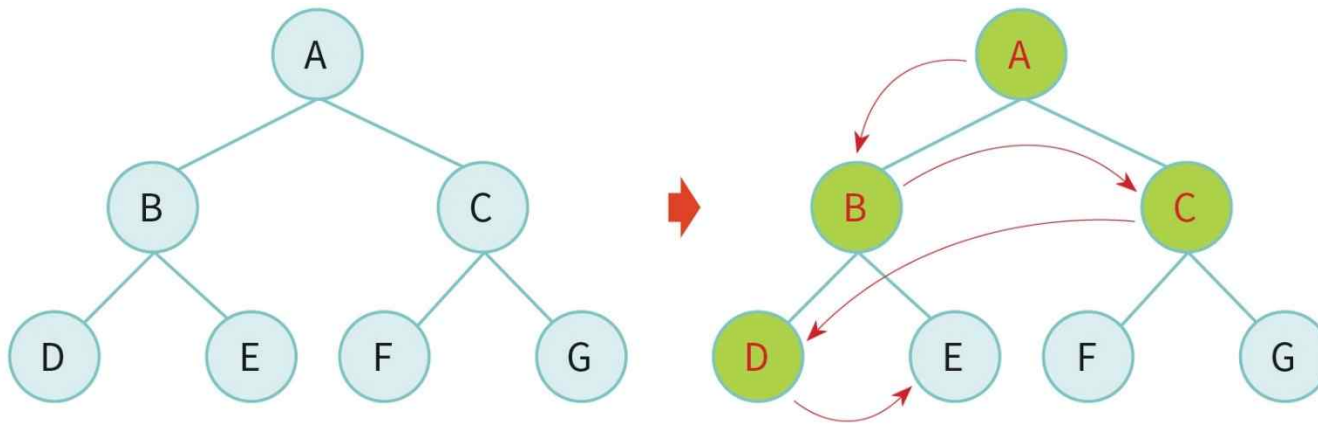
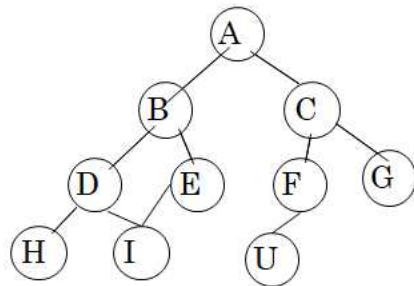


그림 2-13 너비 우선 탐색

BFS 알고리즘

```
function BFS(root)
  open ← [root]
  closed ← []
  while open ≠ [] do
    X ← open 리스트의 첫 번째 요소
    if X == goal then return SUCCESS
    else
      X의 자식 노드를 생성한다.
      X를 closed 리스트에 추가한다.
      X의 자식 노드가 이미 open이나 closed에 있다면 버린다.
      나머지 자식 노드들은 open의 끝에 추가한다. (큐처럼 사용)
  return FAIL
```


BFS 예



1. open = [A]; closed = []
2. open = [B,C]; closed = [A]
3. open = [C,D,E]; closed = [B,A]
4. open = [D,E,F,G]; closed = [C,B,A]
5. open = [E,F,G,H,I]; closed = [D,C,B,A]
6. open = [F,G,H,I]; closed = [E,D,C,B,A] (I는 이미 open 리스트에 있으니 추가되지 않는다.)
7. open = [G,H,I,U]; closed = [F,E,D,C,B,A]
8. open = [H,I,U]; closed = [G,F,E,D,C,B,A]
9. open = [I,U]; closed = [H,G,F,E,D,C,B,A]
10. open = [I,U]; closed = [H,G,F,E,D,C,B,A]
11. open = [U]; closed = [I,H,G,F,E,D,C,B,A]
12. 목표 노드 U 발견!

너비 우선 탐색의 분석

- 완결성: 분기 계수 b 가 유한하면 너비 우선 탐색은 반드시 해답을 발견할 수 있다.
- 시간 복잡도와 공간 복잡도: 일단 생성되는 노드의 개수를 세어보자. 이 노드들이 메모리 상에 있어야 하므로 시간 복잡도와 공간 복잡도는 모두 지수 복잡도 $O(b^d)$ 가 된다. 시간 복잡도보다도 공간 복잡도가 더 문제이다. 분기 계수가 10이고, 깊이 10까지만 탐색한다고 하자. 한 개의 노드를 저장하는데 1KB가 든다면 메모리는 최대 =10TB가 필요하게 된다. 아주 간단한 문제가 아니라면 천문학적인 시간과 메모리 공간이 필요하다고 할 수 있다.

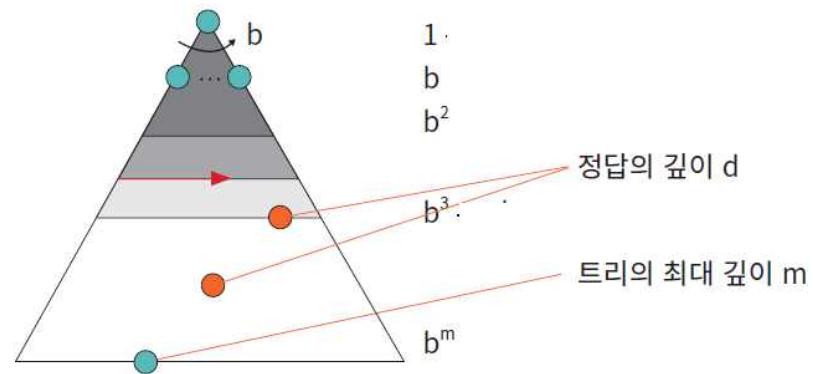
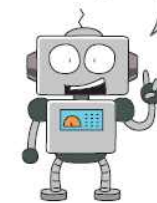


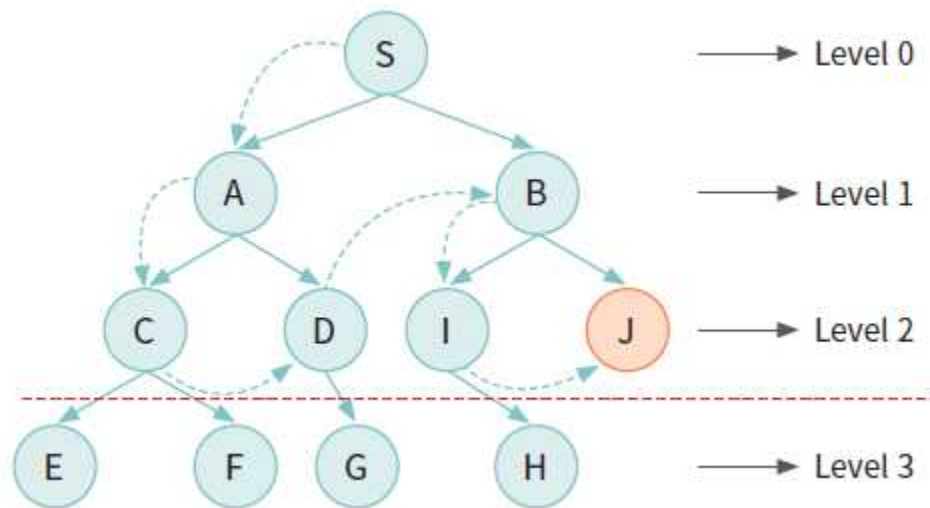
그림 2-18 BFS의 시간과 공간 복잡도



BFS는 가장 가까운 정답을 발견할 수 있습니다. 그러나 메모리가 많이 필요합니다.

깊이 제한 탐색

- 깊이 우선 탐색(**DFS**)은 메모리 소모도 적고 정답을 빠르게 찾을 수도 있지만, 한번 잘못된 경로로 빠지면 나오지 못할 수도 있다. 이 단점을 보완한 것이 깊이를 제한하는 ‘깊이 제한 탐색(**Deep-Limited Search**)’이다. **DFS**와 기본적인 것은 같지만, 끝까지 깊이 들어가지 않고 어떤 한계를 정해서 그 깊이 이상은 탐색하지 않고 백트래킹하는 것이다.



탐색 트리에서 어느 깊이 이상으로 들어가지 못하게 하는 방법입니다.

IDDFS(Iterative Deepening DFS)

- 한계 깊이를 0, 1, 2, 3, 4 ... 이렇게 차례대로 늘려가며 깊이 제한 탐색을 진행하는 것이다. 목표를 찾을 때까지 제한 깊이를 늘려가면서 탐색한다.

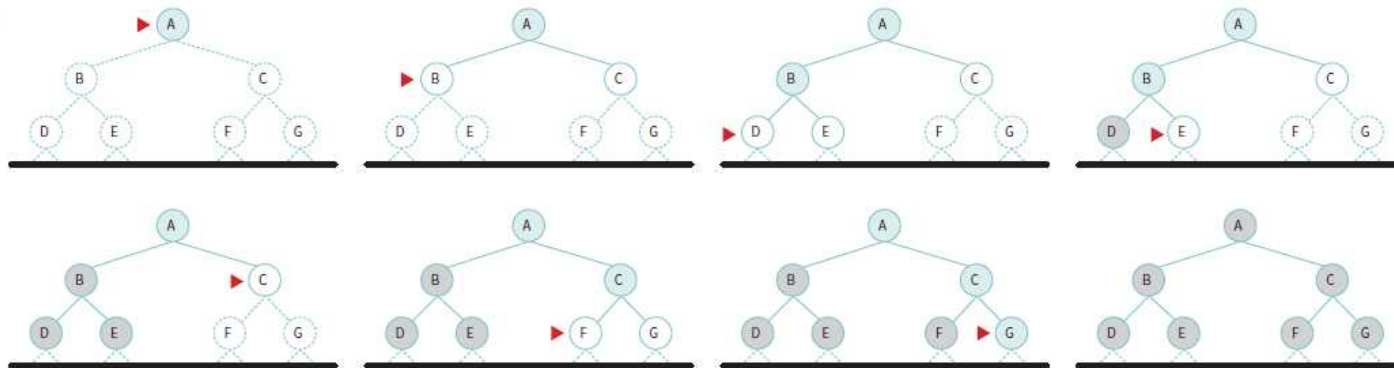
Limit = 0



Limit = 1



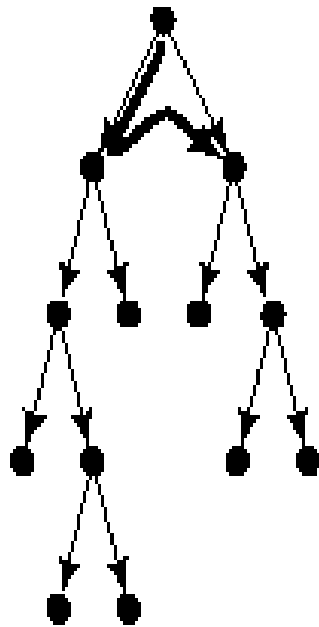
Limit = 2



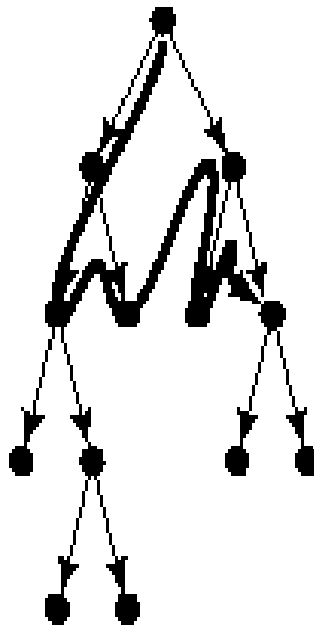
IDDFS 알고리즘

```
function IDDFS(root)
  for depth from 0 to  $\infty$  do
    found, remaining  $\leftarrow$  DFS(root, depth)
    if found  $\neq$  null then
      return found
    else if not remaining then
      return NULL
```

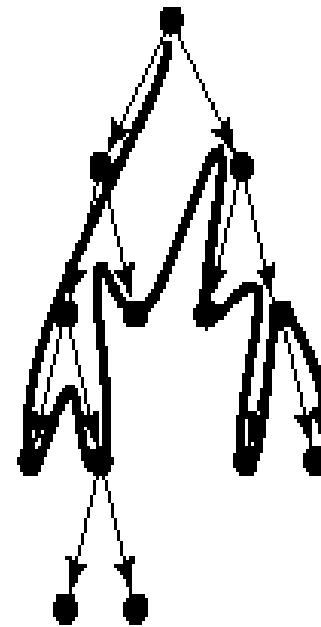
반복적 깊이 증가 (iterative deepening)



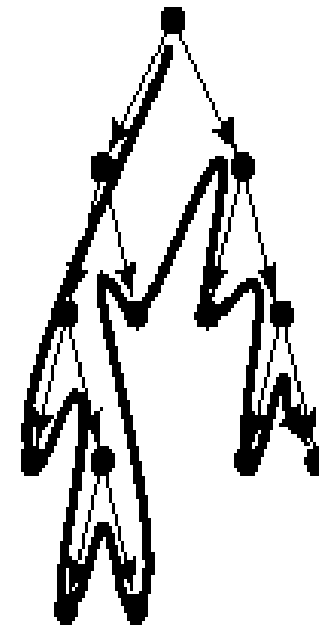
Depth bound = 1



Depth bound = 2



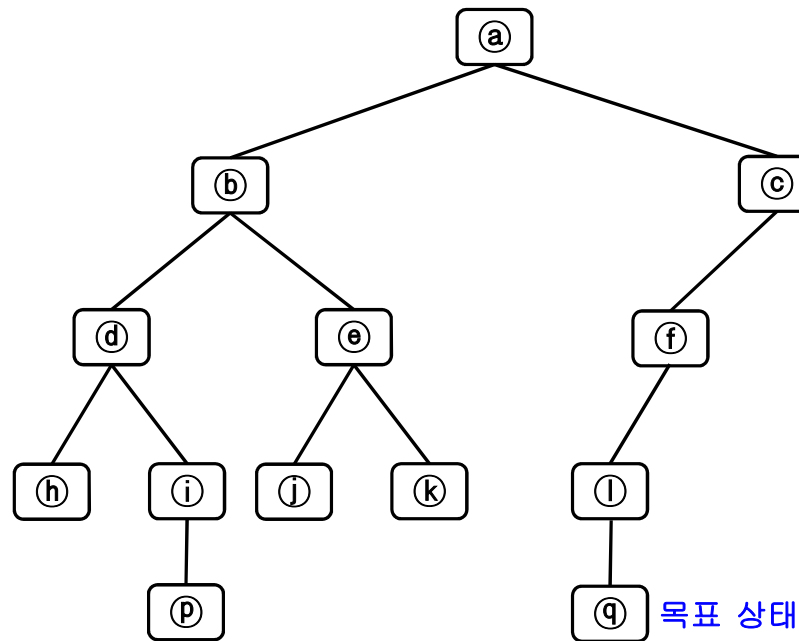
Depth bound = 3



Depth bound = 4

맹목적 탐색

- 반복적 깊이증가 탐색(iterative-deepening search)
 - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



맹목적 탐색

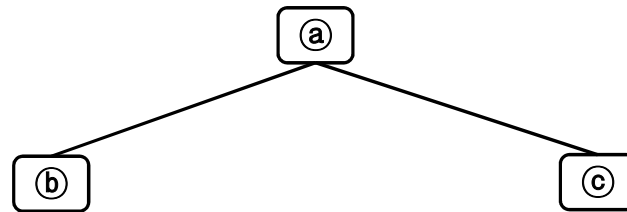
- 반복적 깊이증가 탐색(iterative-deepening search)
 - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



깊이 0: ①

맹목적 탐색

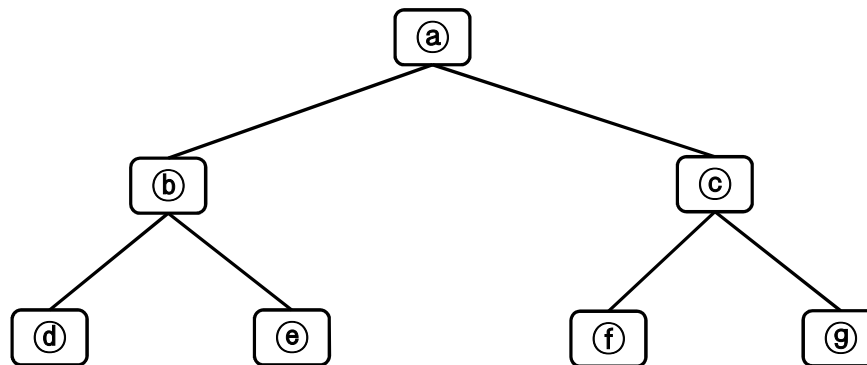
- 반복적 깊이증가 탐색(iterative-deepening search)
 - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



깊이 0: a
깊이 1: a, b, c

맹목적 탐색

- 반복적 깊이증가 탐색(iterative-deepening search)
 - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



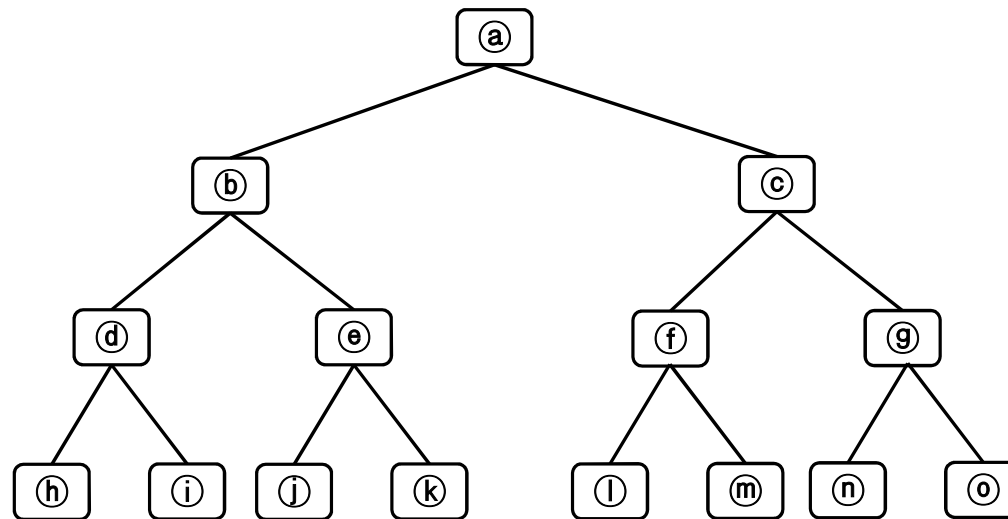
깊이 0: a

깊이 1: a, b, c

깊이 2: a, b, d, e, c, f, g

맹목적 탐색

- 반복적 깊이증가 탐색(iterative-deepening search)
 - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



깊이 0: a

깊이 1: a, b, c

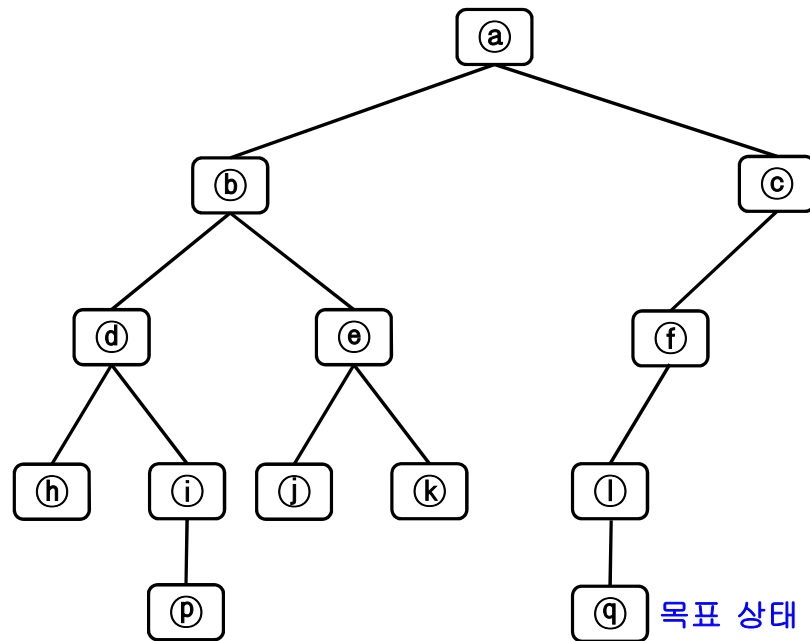
깊이 2: a, b, d, e, c, f, g

깊이 3:

a, b, d, h, i, e, j, k, c, f, l, m, g, n, o

맹목적 탐색

- 반복적 깊이증가 탐색(iterative-deepening search)
 - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



깊이 0: a

깊이 1: a, b, c

깊이 2: a, b, d, e, c, f

깊이 3: a, b, d, h, i, e, j, k, c, f, l

깊이 4: a, b, d, h, i, p, e, j, k, c, f, l, q

반복적 깊이증가 탐색

- 특징 :
 - 깊이 우선 탐색처럼 메모리 필요량이 깊이 제한에 비례 (the linear memory)
 - 최단 경로로 목표 노드를 찾는 것을 보장
 - 목표 노드가 찾아질 때까지 깊이 제한을 1씩 증가
 - 반복적 깊이 증가 탐색에서 확장되는 노드의 수는 너비우선 탐색에서 확장되는 노드의 수보다 그다지 많지 않다
 - 분기 계수가 10이고 목표 노드가 깊이 있는 경우, 너비 우선 탐색에 비해 약 11% 정도 더 노드를 확장

- 분기계수 b , 목표 노드가 깊이 d 에 있는 경우

- 너비우선 탐색으로 확장되는 노드 수

$$N_{bf} = 1 + b + b^2 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

- 깊이 j 까지 깊이 우선 탐색으로 확장되는 노드 수

$$N_{df_j} = \frac{b^{j+1} - 1}{b - 1}$$

- 반복적 깊이증가 탐색

$$\begin{aligned} N_{id} &= \sum_{j=0}^d \frac{b^{j+1} - 1}{b - 1} \\ &= \frac{1}{b - 1} \left[b \left(\sum_{j=0}^d b^j \right) - \sum_{j=0}^d 1 \right] \\ &= \frac{1}{b - 1} \left[b \left(\frac{b^{d+1} - 1}{b - 1} \right) - (d + 1) \right] \\ &= \frac{b^{d+2} - 2b - bd + d + 1}{(b - 1)^2} \end{aligned}$$

$$\frac{N_{id}}{N_{bf}} = \frac{\frac{b^{d+2} - 2b - bd + d + 1}{(b-1)^2}}{\frac{b^{d+1} - 1}{b-1}} = \frac{b^{d+2} - 2b - bd + d + 1}{(b^{d+1} - 1)(b-1)} = \frac{b}{b-1}$$

IDDFS의 장점과 단점

- **IDDFS**는 깊이 우선 탐색의 공간 효율성과 너비 우선 탐색의 완결성을 결합한다. 또 해답이 존재하는 경우, 가장 적은 비용을 갖는 경로를 찾는다. **IDDFS**가 여러 번 동일한 노드를 방문하기 때문에 낭비처럼 보일 수 있지만, 탐색 트리에서는 대부분의 노드가 하위 수준에 있기 때문에 비용이 생각보다 많이 들지 않는다.
- 알고리즘의 응답성이 향상된다. 초기 반복에서는 작은 수의 노드를 사용하기 때문에 매우 빠르게 실행된다. 알고리즘은 초기 결과를 빠르게 제공할 수 있다. 따라서 프로그램이 지금까지 완료한 탐색에서 찾은, 최고의 결과로 언제든지 작업을 수행할 수 있다

DFS와 BFS 프로그램

- 8-puzzle을 BFS 파이썬으로 프로그램 해보자.

```
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
-----
...
...
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
-----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
-----
탐색 성공
```

전체 소스 #1

상태를 나타내는 클래스

```
class State:
```

```
    def __init__(self, board, goal, depth=0):
```

```
        self.board = board
```

```
        self.depth = depth
```

```
        self.goal = goal
```

i1과 i2를 교환하여서 새로운 상태를 반환한다.

```
def get_new_board(self, i1, i2, depth):
```

```
    new_board = self.board[:]
```

```
    new_board[i1], new_board[i2] = new_board[i2], new_board[i1]
```

```
    return State(new_board, self.goal, depth)
```

#shallow copy를 deep copy로 바꿔줘야 함

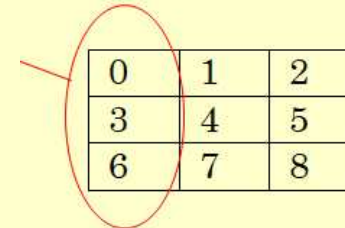
```
import copy
```

```
new_board = copy.deepcopy(self.board[:])
```

전체 소스 #2

자식 노드를 확장하여서 리스트에 저장하여서 반환한다.

```
def expand(self, depth):  
    result = []  
    i = self.board.index(0) # 숫자 0(빈칸)의 위치를 찾는다.  
    if not i in [0, 3, 6]: # LEFT 연산자  
        result.append(self.get_new_board(i, i-1, depth))  
    if not i in [0, 1, 2]: # UP 연산자  
        result.append(self.get_new_board(i, i-3, depth))  
    if not i in [2, 5, 8]: # RIGHT 연산자  
        result.append(self.get_new_board(i, i+1, depth))  
    if not i in [6, 7, 8]: # DOWN 연산자  
        result.append(self.get_new_board(i, i+3, depth))  
    return result
```



| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

전체 소스 #3

객체를 출력할 때 사용한다.

```
def __str__(self):  
    return str(self.board[:3]) + "\n" + \  
    str(self.board[3:6]) + "\n" + \  
    str(self.board[6:]) + "\n" + \  
    "-----"
```

```
def __eq__(self, other):    # 이것을 정의해야 in 연산자가 올바르게 계산한다.  
    return self.board == other.board
```

```
def __ne__(self, other):    # 이것을 정의해야 in 연산자가 올바르게 계산한다.  
    return self.board != other.board
```

초기 상태

```
puzzle = [2, 8, 3,  
          1, 6, 4,  
          7, 0, 5]
```

목표 상태

```
goal = [1, 2, 3,  
        8, 0, 4,  
        7, 6, 5]
```

전체 소스 #4

```
# open 리스트
open_queue = []
open_queue.append(State(puzzle, goal))

closed_queue = []
depth = 0

count=1
while len(open_queue) != 0:
    current = open_queue.pop(0)      # OPEN 리스트의 앞에서 삭제
    print(count)
    count += 1
    print(current)
    if current.board == goal:
        print("탐색 성공")
        break
    depth = current.depth+1
    closed_queue.append(current)
    if depth > 5:
        continue
    for state in current.expand(depth):
        if (state in closed_queue) or (state in open_queue): # 이미 거쳐간 노드이면
            continue      # 노드를 버린다.
        else:
            open_queue.append(state)    # OPEN 리스트의 끝에 추가
```

실행 결과

1

[2, 8, 3]

[1, 6, 4]

[7, 0, 5]

2

[2, 8, 3]

[1, 6, 4]

[0, 7, 5]

...

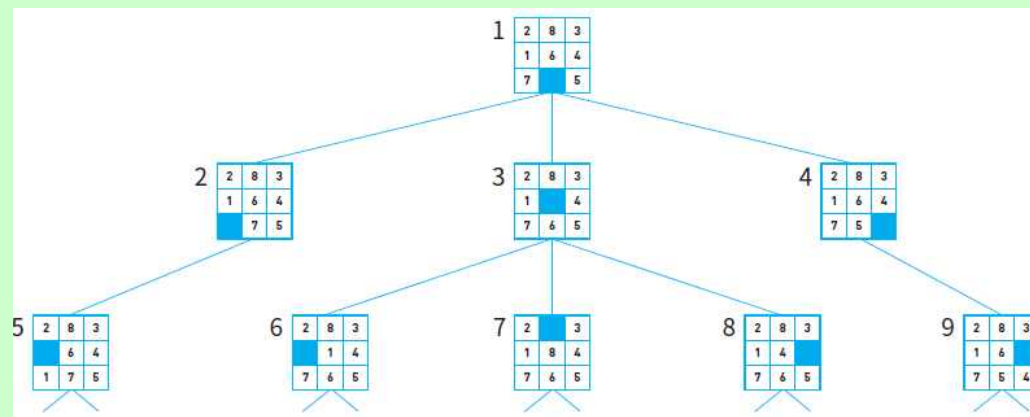
46

[1, 2, 3]

[8, 0, 4]

[7, 6, 5]

탐색 성공



실습

- 앞의 8-퍼즐 프로그램이 최단 경로를 출력하도록 수정하여 실행하십시오.

DFS 프로그램

- DFS 탐색은 무한히 깊이 빠져서 돌아 오지 못할 수도 있다. 따라서 우리는 깊이를 제한하는 것이 필요하다. 소스는 다음과 같이 변경된다.

자식 노드를 확장하여서 리스트에 저장하여서 반환한다.

```
def expand(self, depth):
```

```
    result = []
```

```
    if depth > 5: return result # 깊이가 5 이상이면 더 이상 확장을 하지 않는다.
```

```
    i = self.board.index(0) # 숫자 0(빈칸)의 위치를 찾는다.
```

```
    if not i in [6, 7, 8]: # DOWN 연산자
```

```
        result.append(self.get_new_board(i, i+3, moves))
```

```
    if not i in [2, 5, 8]: # RIGHT 연산자
```

```
        result.append(self.get_new_board(i, i+1, moves))
```

```
    if not i in [0, 1, 2]: # UP 연산자
```

```
        result.append(self.get_new_board(i, i-3, moves))
```

```
    if not i in [0, 3, 6]: # LEFT 연산자
```

```
        result.append(self.get_new_board(i, i-1, moves))
```

```
    return result
```

실행 결과

1

[2, 8, 3]

[1, 6, 4]

[7, 0, 5]

...

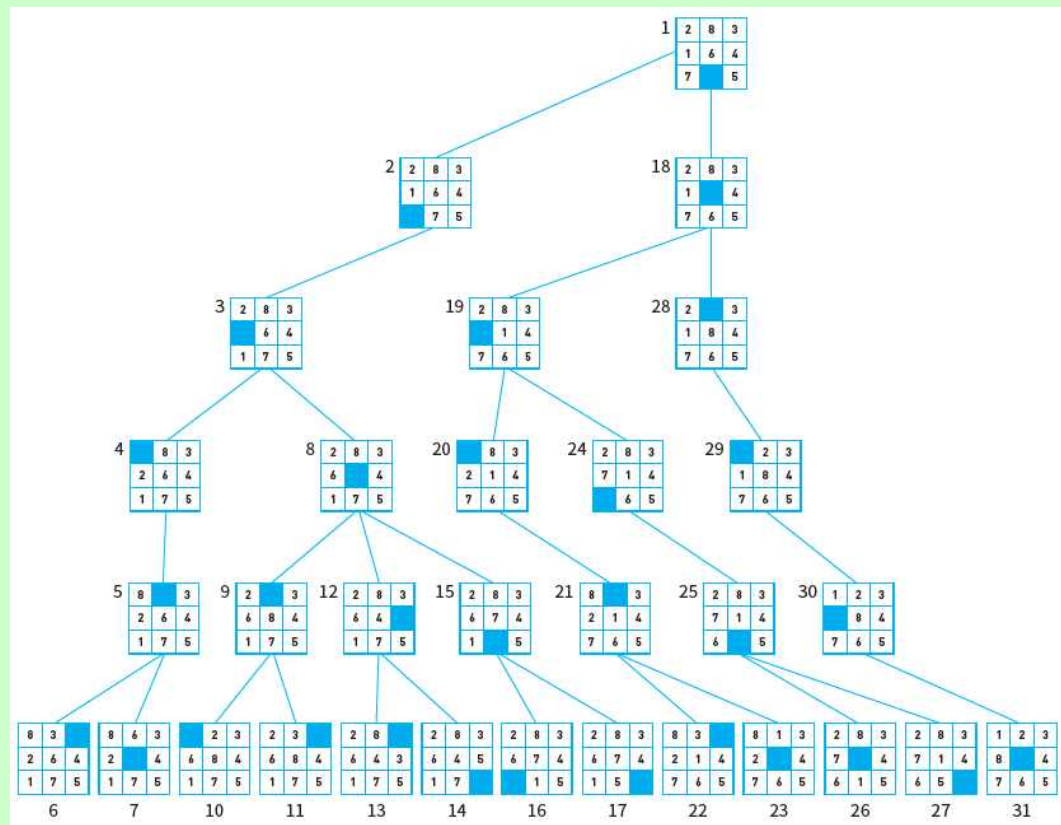
31

[1, 2, 3]

[8, 0, 4]

[7, 6, 5]

탐색 성공

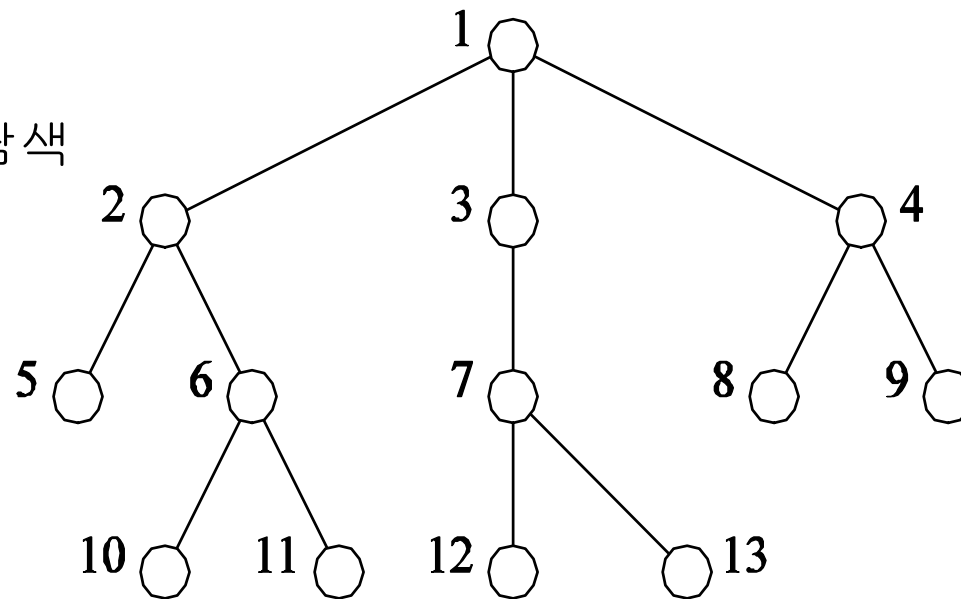


맹목적 탐색

- 맹목적 탐색 방법의 비교
 - 깊이 우선 탐색
 - 메모리 공간 사용 효율적
 - 최단 경로 해 탐색 보장 불가
 - 너비 우선 탐색
 - 최단 경로 해 탐색 보장
 - 메모리 공간 사용 비효율
 - 반복적 깊이증가 탐색
 - 최단 경로 해 보장
 - 메모리 공간 사용 효율적
 - 반복적인 깊이 증가 탐색에 따른 비효율성
 - 실제 비용이 크게 늘지 않음
 - 각 노드가 10개의 자식노드를 가질 때, 너비 우선 탐색 대비 약 11%정도 추가 노드 생성
 - 맹목적 탐색 적용시 우선 고려 대상

- 다음과 같은 트리가 있다고 하자. 노드 1에서 시작하여 아래 지정한 탐색을 시작할 때, 방문하는 노드들의 기호를 순서대로 쓰시오. 단, 자식 노드들은 왼쪽에 있는 것부터 탐색을 한다고 가정한다.

1. 너비우선 탐색
2. 깊이우선 탐색
3. 반복적 깊이증가 탐색



실습: 반복적 깊이 증가 탐색 구현

- 8-puzzle을 반복적 깊이 증가 탐색으로 해결하고 최적 경로를 출력하는 파이썬 프로그램을 구현하시오.

```
function IDDFS(root)
  for depth from 0 to  $\infty$  do
    found, remaining  $\leftarrow$  DFS(root, depth)
    if found  $\neq$  null then
      return found
    else if not remaining then
      return NULL
```

8 경험적인 탐색 방법

- 만약 우리가 문제 영역에 대한 정보나 지식을 사용할 수 있다면 탐색 작업을 훨씬 빠르게 할 수 있다. 이것을 경험적 탐색 방법(**heuristic search method**) 또는 휴리스틱 탐색 방법이라고 부른다.
- 이때 사용되는 정보를 휴리스틱 정보(**heuristic information**)라고 한다.



8-puzzle에서의 휴리스틱

예를 들어서 현재 상태와 목표 상태가 다음과 같다고 하자.

| | | |
|---|---|---|
| 2 | 1 | 3 |
| 8 | 5 | 6 |
| 7 | 4 | |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

▷ $h1(N)$ = 현재 제 위치에 있지 않은 타일의 개수 = $1+1+1+1=4$

| | | |
|---|---|---|
| 2 | 1 | 3 |
| 8 | 5 | 6 |
| 7 | 4 | |

▷ $h2(N)$ = 각 타일의 목표 위치까지의 거리 = $1+1+0+2+0+0+0+2=6$

| | | |
|---|---|---|
| 2 | 1 | 3 |
| 8 | 5 | 6 |
| 7 | 4 | |

09 언덕 등반 기법

- 이 기법에서는 평가 함수의 값이 좋은 노드를 먼저 처리한다.
- 평가함수로 $h1(N)$, 제 위치에 있지 않은 타일의 개수 사용

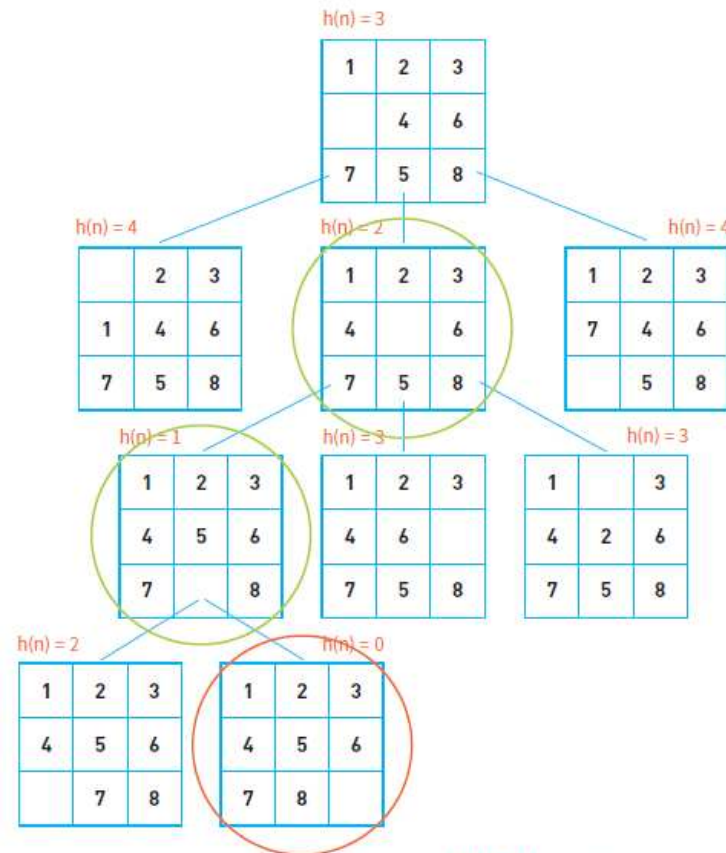


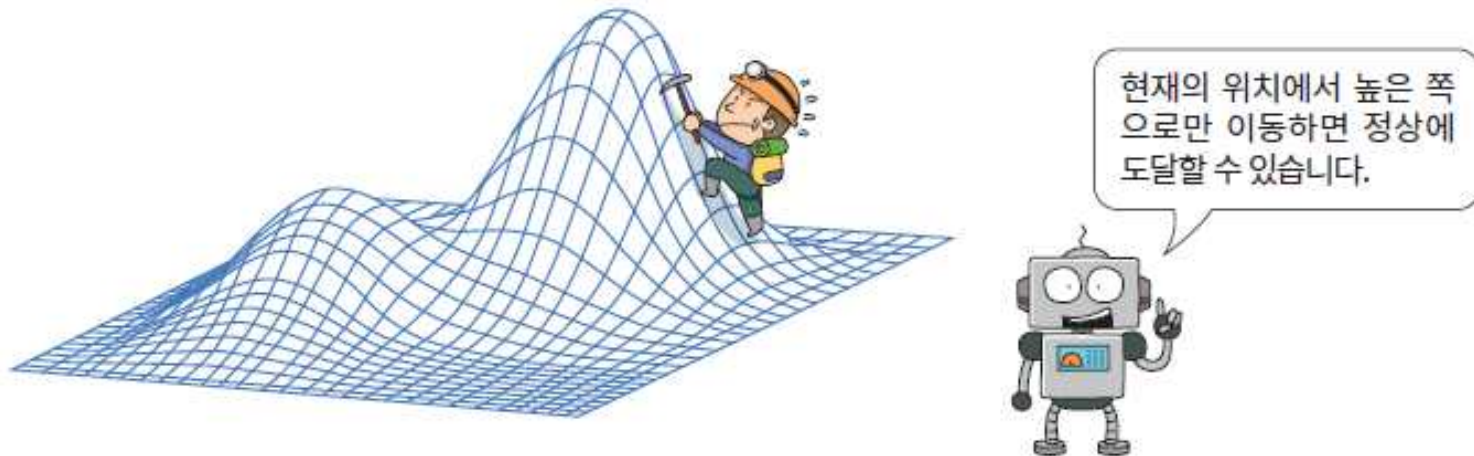
그림 2-14 언덕 등반 기법



탐색 트리에서 평가 함수 값이 좋은 노드만 선택하는 방법입니다.

언덕 등반 기법

- 탐욕적인(언덕 등반) 탐색 방법은 무조건 휴리스틱 함수 값이 가장 좋은 노드만을 선택한다. 이전 기록을 보관하지 않음.
- 이것은 등산할 때 무조건 현재의 위치보다 높은 위치로만 이동하는 것과 같다. 일반적으로는 현재의 위치보다 높은 위치로 이동하면 산의 정상에 도달할 수 있다.



언덕 등반 기법 알고리즘

1. 먼저 현재 위치를 기준으로 해서, 각 방향의 높이를 판단한다.(노드의 확장)
2. 만일 모든 위치가 현 위치보다 낮다면 그 곳을 정상이라고 판단한다(목표상태인가의 검사).
3. 현 위치가 정상이 아니라면 확인된 위치 중 가장 높은 곳으로 이동한다(후계노드의 선택).

언덕 등반 기법 알고리즘

```
function HILL_CLIMBING(root)
```

```
  current ← root
```

```
  loop do
```

```
    현재 노드의 자식 노드들을 생성한다.
```

```
    neighbor ← 평가 함수값이 가장 높은 자식 노드
```

```
    # 만일 모든 위치가 현 위치보다 낮다면 그 곳을 정상이라고 판단한다
```

```
    if VALUE(neighbour) ≤ VALUE(current) then
```

```
      return current
```

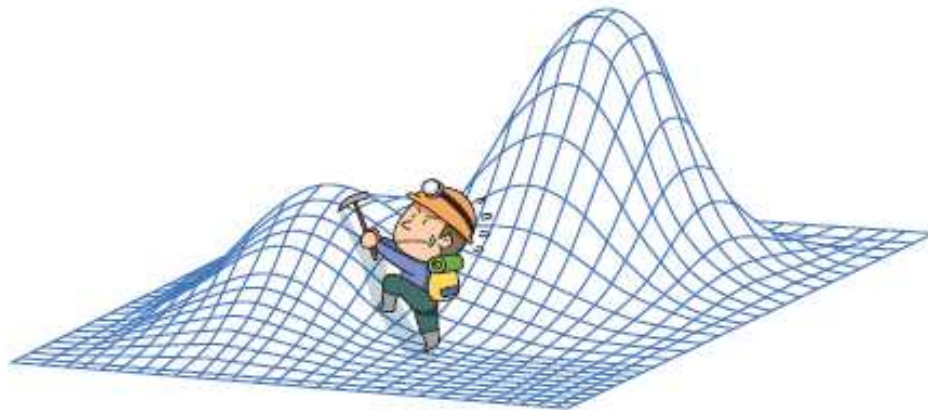
```
    # 현 위치가 정상이 아니라면 확인된 위치 중 가장 높은 곳으로 이동한다.
```

```
    current ← neighbor
```

언덕 등반 기법에서는
OPEN과 CLOSED
리스트를 사용하지 않는다.
LOL.

지역 최대 문제

- 순수한 언덕 등반 기법은 오직 $h(n)$ 값만을 사용한다(**OPEN** 리스트나 **CLOSED** 리스트를 사용하지 않는다 -> 과거 기억이 없음).
- 이런 경우에는 생성된 자식 노드의 평가함수 값이 부모 노드보다 더 좋지 않은 경우가 나올 수 있다. 이것을 지역 최대 문제(local maximum problem)라고 한다.



지역 최대값에 도달하면 아무리 노력을 해도 전역 최대값으로 가지 못하는 문제입니다. 언덕 등반 기법에서는 지역 최대값만 찾을 수 있습니다.

지역 최대 문제의 예

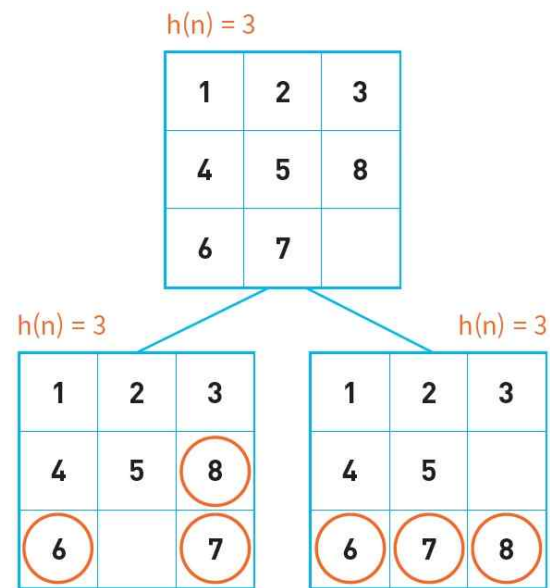


그림 2-15 지역 최대 문제

11 최고 우선 탐색

- 최고 우선 탐색 (Best-First Search)
 - 언덕 등반 기법을 개선한 것
 - open과 closed 리스트를 사용
 - open 리스트에서 다음에 처리할 노드를 선택
 - 가장 유망한 노드(평가 함수 값이 가장 좋은 노드)를 선택
 - 지역 최대값에 있더라도 내려갈 수 있음
 - open에서 가장 좋은 노드가 기존 지역 최대보다 안 좋을 수 있음
 - 더 안 좋은 길로 이동하여 지역최대를 벗어날 수 있음

최고 우선 탐색 알고리즘

```
function BEST_FIRST(root)
  open ← [root]
  closed ← []
  while open ≠ [] do
    X ← open 리스트에서 가장 평가 함수의 값이 좋은 노드
    if X == goal then return SUCCESS
    else
      X의 자식 노드를 생성한다.
      X를 closed 리스트에 추가한다.
      if X의 자식 노드가 이미 open이나 close에 있지 않으면
        자식 노드의 평가 함수값을 계산한다.
        자식 노드들을 open 리스트에 추가한다.
  return FAIL
```

휴리스틱 탐색

- 평가 함수를 사용

- 문제의 특성에 대한 정보인 휴리스틱에 따라 목표까지의 가장 좋은 경로상에 있다고 판단되는 노드를 우선 방문
- open과 closed 리스트를 사용(언덕 등반 기법과 다른점)

- 최상우선 탐색(*Best-First Search*)

1. 다음에 어느 노드를 확장하는 것이 최선인지를 결정하는 데 도움이 되는 휴리스틱 평가 함수(*evaluation function*) \hat{f} 이 있다고 가정: 각 상태 표현에 대해 실수값을 가짐
2. open 리스트에서 $\hat{f}(n)$ 값이 가장 작은 노드를 다음에 확장할 노드로 선택. 같은 값을 갖는 노드들은 무작위로 선택
3. 다음에 확장할 노드가 목표 노드이면 탐색을 종료

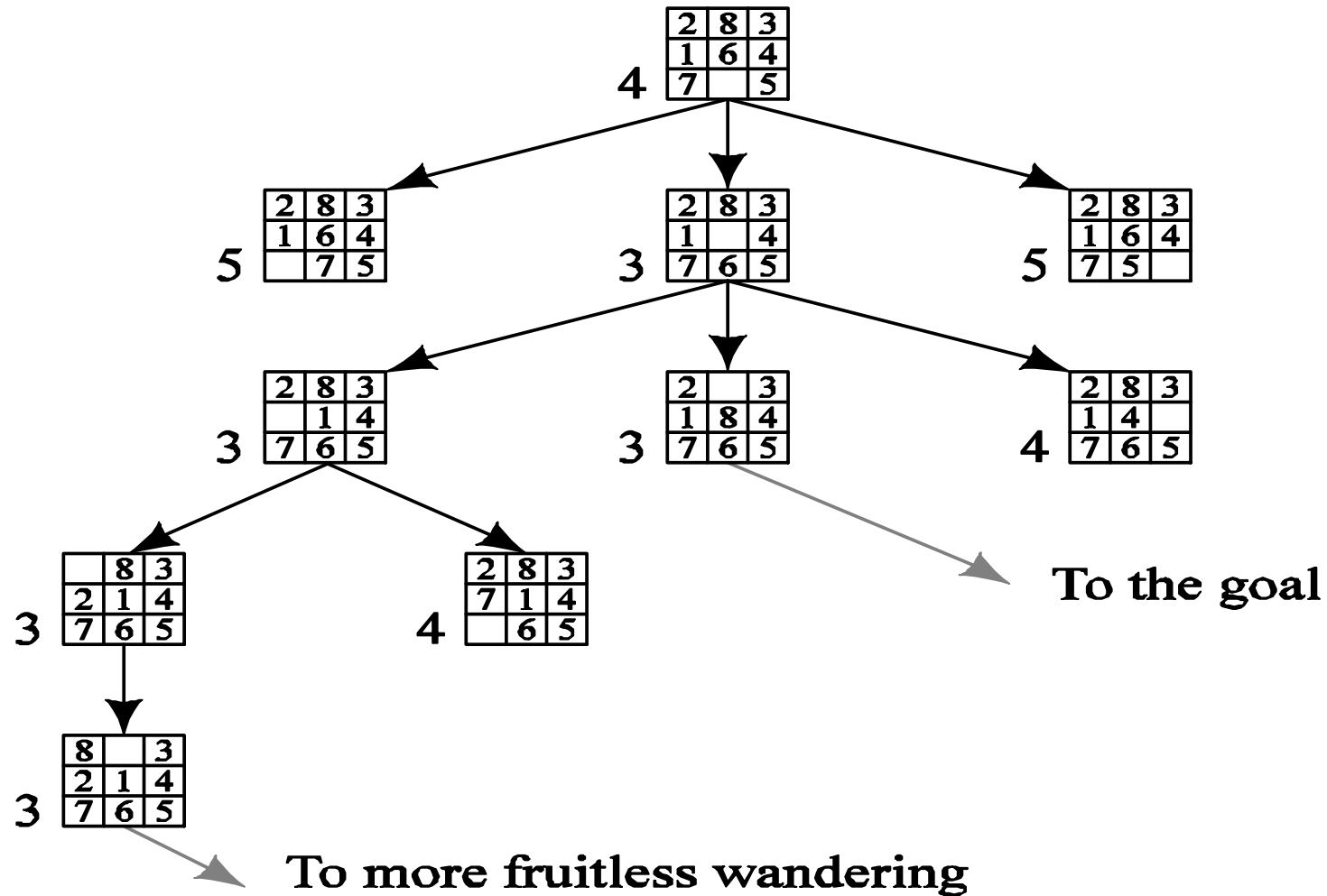
(ex) 8 퍼즐

$\hat{f}(n)$ = 목표상태와 비교해서 제자리에 있지 않은 타일의 개수

휴리스틱 탐색

8 퍼즐에 아래 휴리스틱 함수를 적용

$\hat{f}(n)$ = 목표상태와 비교해서 제자리에 있지 않은 타일의 개수



휴리스틱 탐색

- 탐색 과정이 일찍 만들어진 노드를 선호하도록 할 필요가 있다는 것을 보여주는 예였음
 - ▣ 지나치게 낙관적인 휴리스틱에 의해 잘못된 길로 계속 내려가는 것을 막기 위해서

- \hat{f} 에 깊이요소를 추가

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

$\hat{g}(n)$: 그래프상에서 n 의 깊이에 대한 추정값

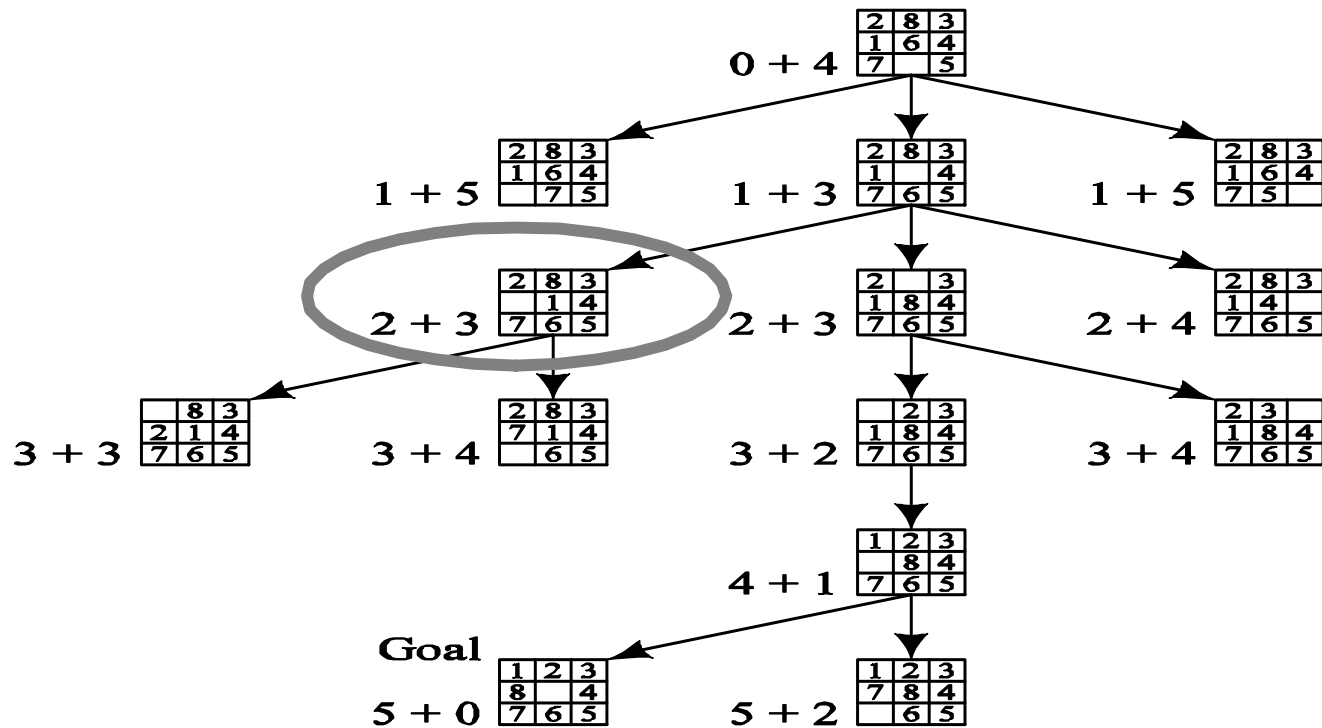
$\hat{h}(n)$: 노드 n 에 대한 휴리스틱 평가값

Heuristic Search Using

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

$\hat{g}(n)$ = n의 깊이에 대한 추정값

$\hat{h}(n)$ = 노드 n에 대한 휴리스틱 평가값



11 A* 알고리즘

- A* 알고리즘은 평가 함수의 값을 다음과 같은 수식으로 정의한다.

$$f(n) = g(n) + h(n)$$

- $g(n)$: 시작 노드에서 현재 노드까지의 비용
- $h(n)$: 현재 노드에서 목표 노드까지의 거리

8-puzzle 에서의 A* 알고리즘

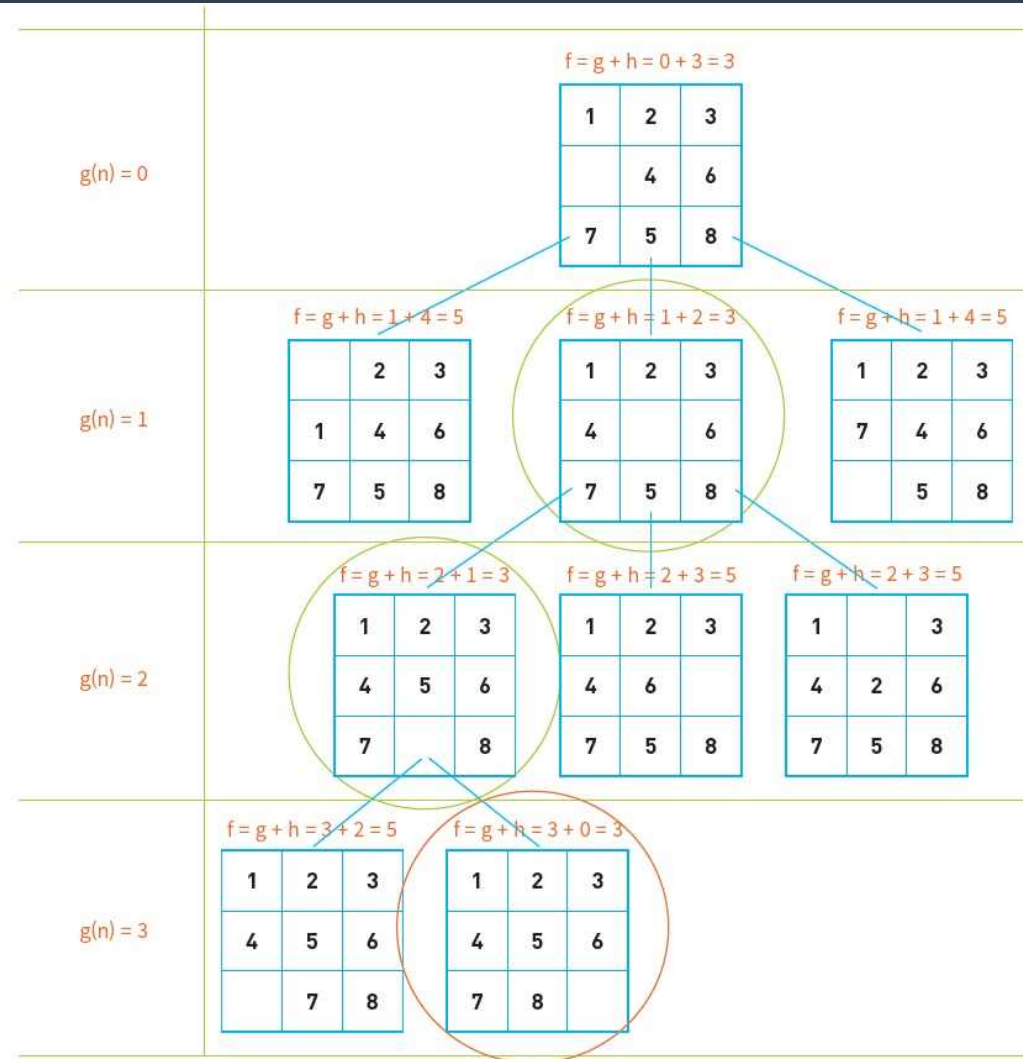


그림 2-17 8-퍼즐에서의 A* 알고리즘

A* 알고리즘

AStar_search()

open \leftarrow [시작노드]

closed \leftarrow []

while open \neq [] do

 X \leftarrow open 리스트에서 평가 함수의 값이 가장 좋은 노드

 if X == goal then return SUCCESS

 else

 X의 자식 노드를 생성한다.

 X를 closed 리스트에 추가한다.

 if X의 자식 노드가 open이나 closed에 있지 않으면

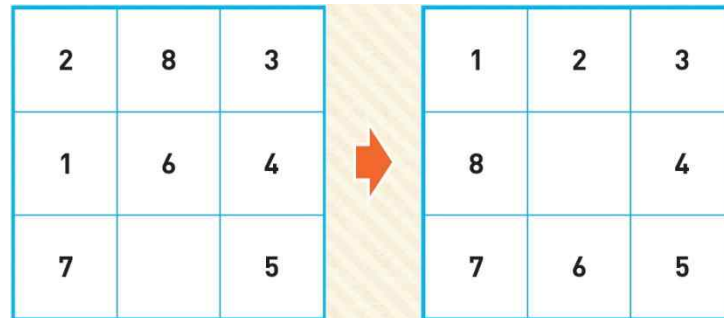
 자식 노드의 평가 함수 값 $f(n) = g(n) + h(n)$ 을 계산한다.

 자식 노드들을 open에 추가한다.

return FAIL

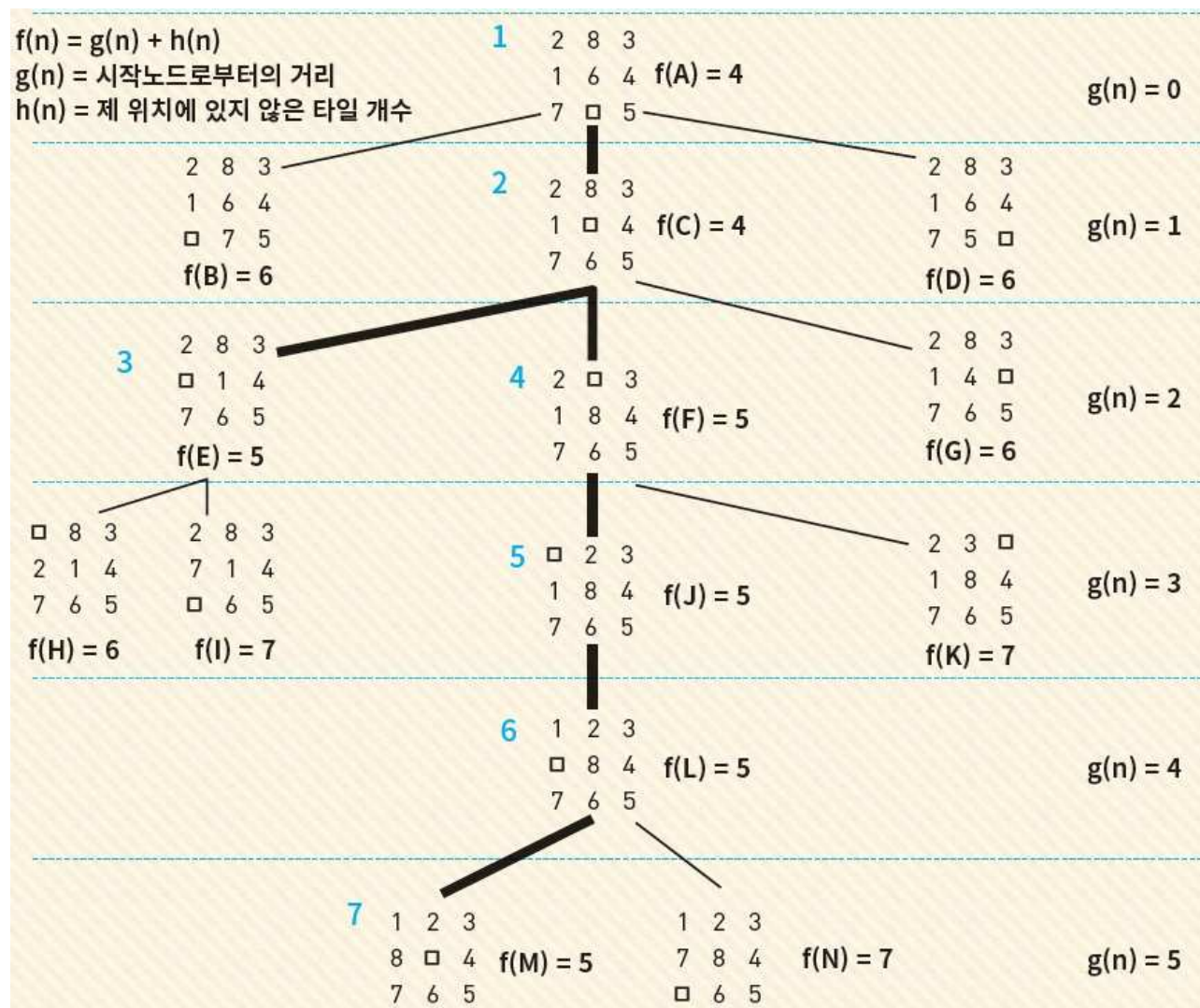
Lab: A* 알고리즘을 시뮬레이션

- 시작 상태와 목표 상태



- $f(n)=g(n)+h(n)$ 이라고 하고 $h(n)$ 은 제 위치에 있지 않은 타일의 개수

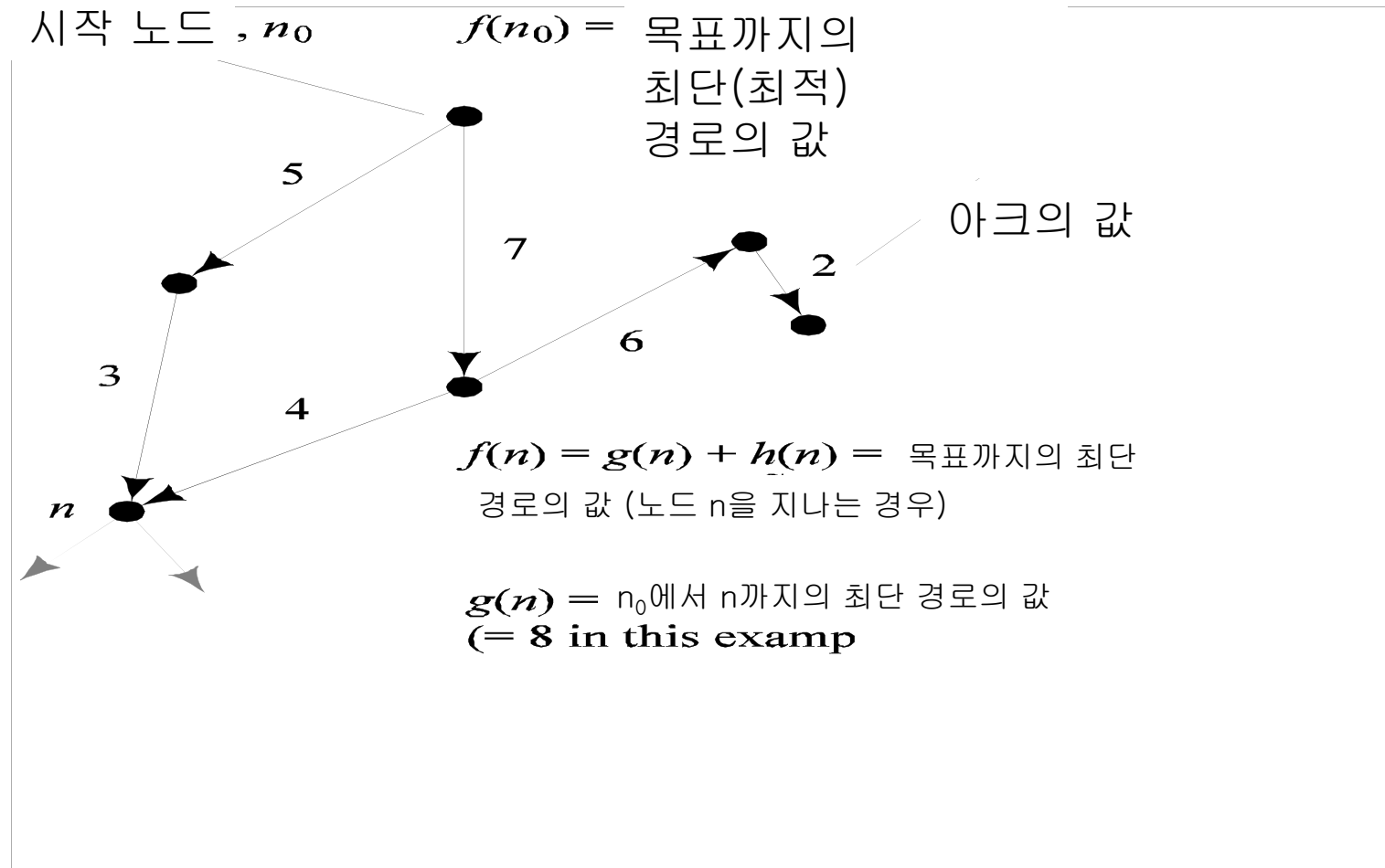
Lab: A* 알고리즘을 시뮬레이션



A* 알고리즘

- $g(n)$: 시작 노드 n_0 에서 노드 n 까지의 최단 경로의 값, $\hat{g}(n)$ 는 A*에 의하여 지금까지 발견된 노드 n 까지의 경로 중에서 최단 경로의 값
- $h(n)$: 노드 n 과 목표 노드 사이의 최단 경로의 실제 값, $\hat{h}(n)$ 은 $h(n)$ 의 추정값
- $f(n) = g(n) + h(n)$
 - ▣ n_0 에서 목표 노드까지 노드 n 을 통하여 갈 수 있는 모든 가능한 경로 중에서 최단 경로의 값
 - ▣ $f(n_0) = h(n_0)$ 는 n_0 에서 목표 노드까지의 최단 경로의 값
- A*에서는 $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$ 을 평가 함수로 사용
 - ▣ \hat{h} 이 0이면 균일비용 탐색

A* 알고리즘



A*의 허용성(Admissibility)

A*가 항상 최단 경로를 찾는 것을 보장하는 3가지 조건:

1. 그래프의 각 노드는 유한개의 자식 노드를 가진다.
2. 그래프의 모든 아크는 임의의 양수 ϵ 보다 큰 값을 갖는다.
3. 탐색 그래프의 모든 노드 n 에 대하여,

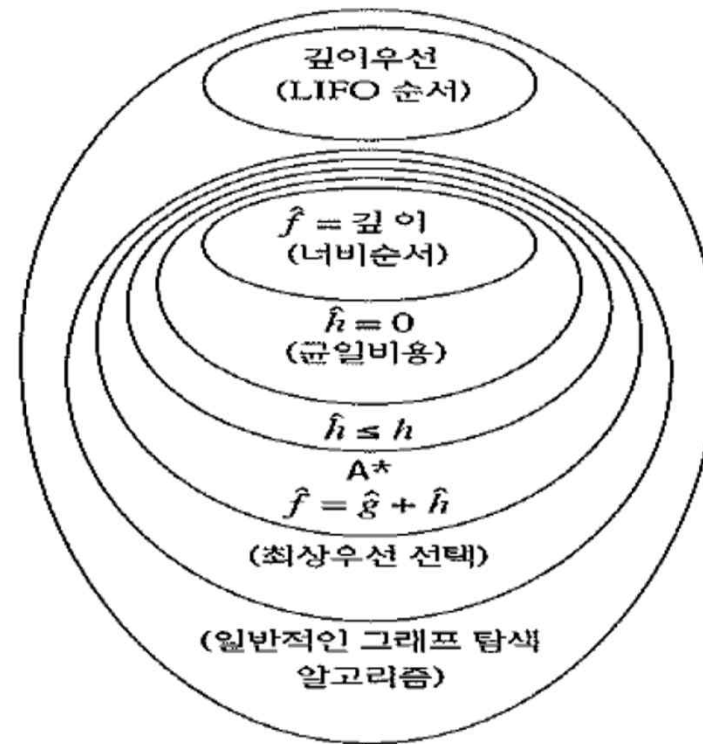
$$\hat{h}(n) \leq h(n)$$

이러한 세 개의 조건을 만족하면 알고리즘 A* 는 목표까지의 경로가 있
기만 하면 항상 최적 경로(optimal path)를 찾는다는 것을 보장한다.
=> 허용 가능(Admissible): 목표까지의 최적 경로를 찾는 것을 보장

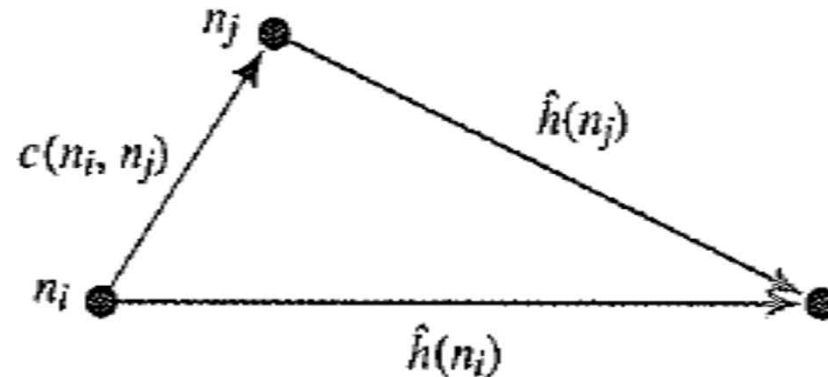
두 개의 서로 다른 A* 알고리즘, A*₁와 A*₂가 , 목표 외의 모든 노드에
대하여 $\hat{h}_1 < \hat{h}_2$ 인 점만 다르다면, A*₂ 가 A*₁보다 정보가 많다
(more informed)고 한다.

A*₂가 A*₁보다 정보가 많으면(more informed), n_0 에서 목표 노드까지
경로가 있는 모든 그래프에 대해서, 탐색이 종료되었을 때 A*₂에 의
해 확장된 노드는 A*₁에 의해서도 확장된다.

탐색 알고리즘 사이의 관계



A*의 일관성(단조성) 조건



$$\hat{h}(n_i) \leq c(n_i, n_j) + \hat{h}(n_j)$$

모든 노드 쌍이 위의 조건을 만족하면 **일관성 조건**을 만족한다고 함

A*의 일관성 조건 = A*의 단조성 조건

\hat{f} 값이 시작 노드에서 멀어짐에 따라 단조적으로 증가함: $\hat{f}(n_j) \geq \hat{f}(n_i)$

일관성 조건(consistency condition) 이 만족되면, A*가 어떤 노드 n 을 확장하면 n 까지의 최적 경로가 이미 발견된 것이다.

A*의 일관성(단조성) 조건

$$\hat{h}(n_i) \leq \hat{h}(n_j) + c(n_i, n_j)$$

$$\hat{h}(n_i) - c(n_i, n_j) \leq \hat{h}(n_j)$$

$$\hat{h}(n_i) - c(n_i, n_j) \leq \hat{h}(n_j) \quad \text{양변에 } \hat{g}(n_j) \text{를 더하면}$$

$$\hat{h}(n_i) + \hat{g}(n_j) - c(n_i, n_j) \leq \hat{h}(n_j) + \hat{g}(n_j)$$

$$\text{여기서 } \hat{g}(n_j) = \hat{g}(n_i) + c(n_i, n_j) \text{ 이므로}$$

$$\hat{h}(n_i) + \hat{g}(n_i) \leq \hat{h}(n_j) + \hat{g}(n_j)$$

$$\hat{f}(n_i) \leq \hat{f}(n_j)$$

A* 알고리즘 파이썬 구현

```
import queue

# 상태를 나타내는 클래스, f(n) 값을 저장한다.
class State:
    def __init__(self, board, goal, depth=0):
        self.board = board      # 현재의 보드 상태
        self.depth = depth      # 깊이
        self.goal = goal        # 목표 상태

    # i1과 i2를 교환하여서 새로운 상태를 반환한다.
    def get_new_board(self, i1, i2, depth):
        new_board = self.board[:]
        new_board[i1], new_board[i2] = new_board[i2], new_board[i1]
        return State(new_board, self.goal, depth)
```

A* 알고리즘 파이썬 구현

자식 노드를 확장하여서 리스트에 저장하여서 반환한다.

```
def expand(self, moves):
```

```
    result = []
```

```
    i = self.board.index(0)    # 숫자 0(빈칸)의 위치를 찾는다.
```

```
    if not i in [0, 3, 6]:    # LEFT 연산자
```

```
        result.append(self.get_new_board(i, i-1, moves))
```

```
    if not i in [0, 1, 2]:    # UP 연산자
```

```
        result.append(self.get_new_board(i, i-3, moves))
```

```
    if not i in [2, 5, 8]:    # RIGHT 연산자
```

```
        result.append(self.get_new_board(i, i+1, moves))
```

```
    if not i in [6, 7, 8]:    # DOWN 연산자
```

```
        result.append(self.get_new_board(i, i+3, moves))
```

```
    return result
```


A* 알고리즘 파이썬 구현

f(n)을 계산하여 반환한다.

```
def f(self):
```

```
    return self.h()+self.g()
```

휴리스틱 함수 값인 h(n)을 계산하여 반환한다.

현재 제 위치에 있지 않은 타일의 개수를 리스트 함축으로 계산한다.

```
def h(self):
```

```
    score = 0
```

```
    for i in range(9):
```

```
        if self.board[i]!=0 and self.board[i] != self.goal[i]:
```

```
            score += 1
```

```
    return score
```

시작 노드로부터의 깊이를 반환한다.

```
def g(self):
```

```
    return self.depth
```

A* 알고리즘 파이썬 구현

```
def __eq__(self, other):  
    return self.board == other.board  
  
def __ne__(self, other):  
    return self.board != other.board  
  
# 상태와 상태를 비교하기 위하여 less than 연산자를 정의한다.  
def __lt__(self, other):  
    return self.f() < other.f()  
  
def __gt__(self, other):  
    return self.f() > other.f()  
  
# 객체를 출력할 때 사용한다.  
def __str__(self):  
    return f"f(n)={self.f()} h(n)={self.h()} g(n)={self.g()}\n"+\  
        str(self.board[:3]) + "\n"+\  
        str(self.board[3:6]) + "\n"+\  
        str(self.board[6:]) + "\n"
```

A* 알고리즘 파이썬 구현

초기 상태

```
puzzle = [2, 8, 3,  
          1, 6, 4,  
          7, 0, 5]
```

목표 상태

```
goal = [1, 2, 3,  
        8, 0, 4,  
        7, 6, 5]
```

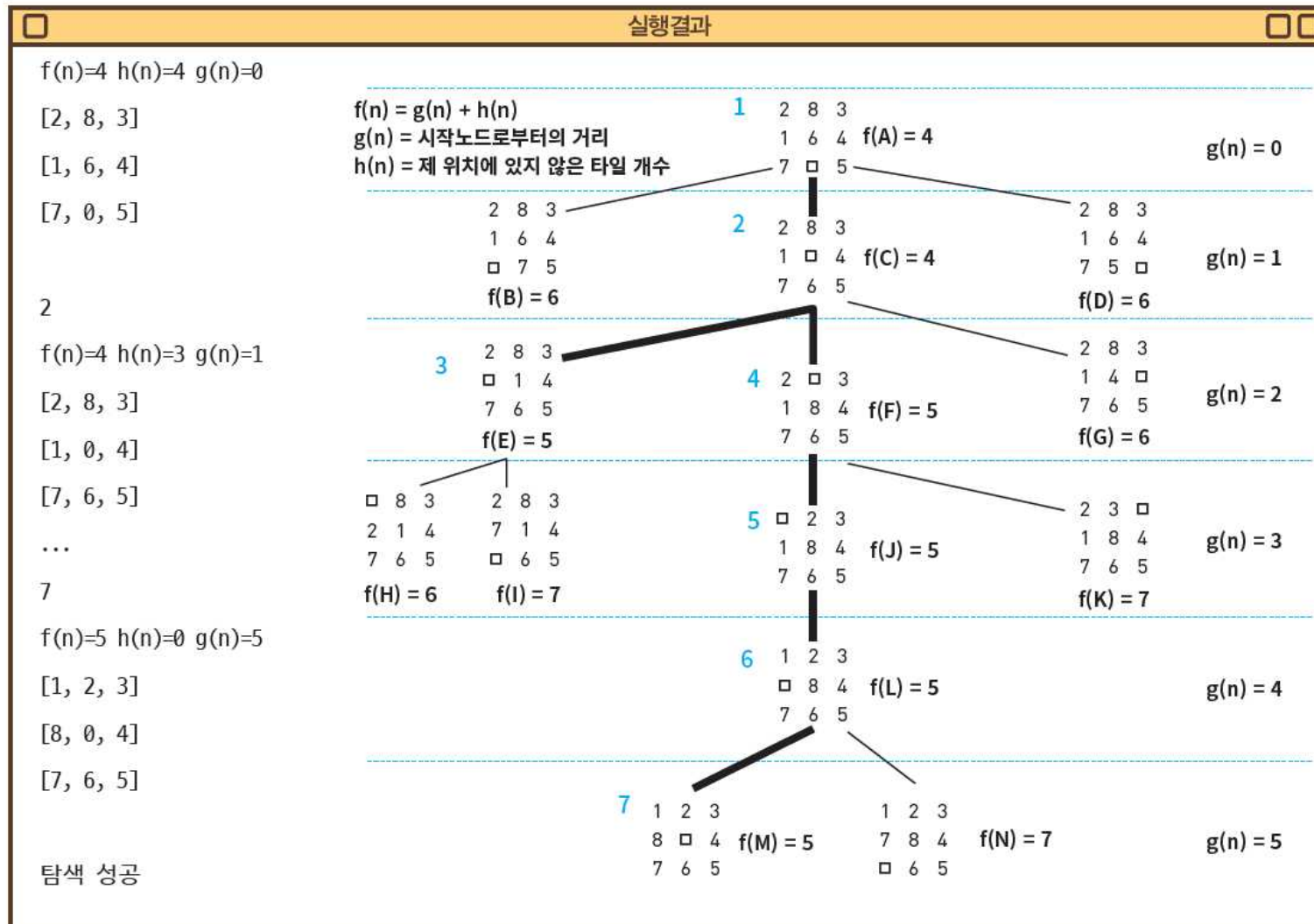
open 리스트는 우선순위 큐로 생성한다.

```
open_queue = queue.PriorityQueue()  
open_queue.put(State(puzzle, goal))
```

A* 알고리즘 파이썬 구현

```
closed_queue = [ ]
depth = 0
count = 0
while not open_queue.empty():
    current = open_queue.get()
    count += 1
    print(count)
    print(current)
    if current.board == goal:
        print("탐색 성공")
        break
    depth = current.depth+1
    for state in current.expand(depth):
        if state not in closed_queue and state not in open_queue.queue :
            open_queue.put(state)
    closed_queue.append(current)
else:
    print ('탐색 실패')
```

실행 결과

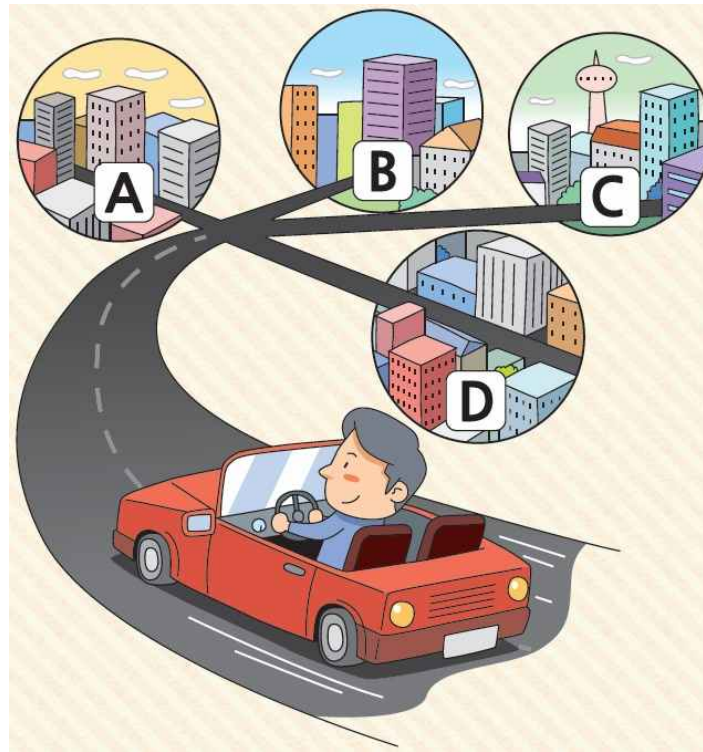


실습

- 앞의 프로그램을 찾아낸 최단 경로를 출력하도록 수정하시오.
- 휴리스틱 함수를 **h2()**로 변경하여 실행하시오.
 - **h2()**: 각 타일의 목표 위치까지의 거리의 합

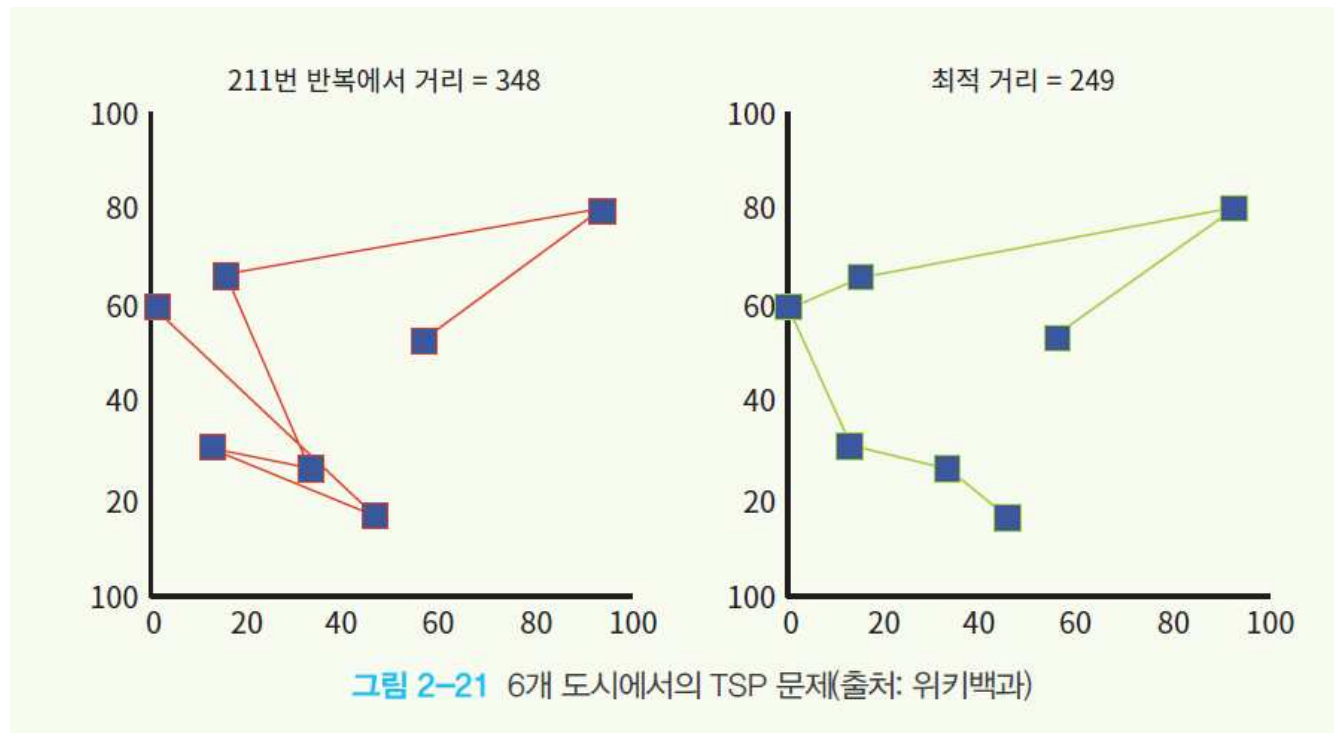
Lab: TSP

- TSP(travelling salesman problem)은 "도시의 목록과 도시들 사이의 거리가 주어졌을 때, 하나의 도시에서 출발하여 각 도시를 방문하는 최단 경로는 무엇인가?" 이다.



Lab: TSP

- 6개의 도시가 있고 6개의 도시를 어떤 순서대로 방문해야 최단 경로가 되는지를 왼쪽에서 계속 계산한다. 오른쪽은 현재까지 발견된 최단 경로이다.




Lab: TSP

- **TSP에서의 휴리스틱:** 아직 방문하지 않은 도시 중에서 가장 가까운 도시를 다음 방문 도시로 선택할 수 있다. 이것을 최근접 휴리스틱 (**Nearest Neighbor Heuristic**)이라고 한다. 이 알고리즘은 효과적으로 짧은 경로를 비교적 신속하게 산출한다.
 1. 아무 도시나 시작점으로 선택하고, 경로에 추가한다.
 2. 현재 경로의 마지막 도시에서 가장 가까운 도시를 찾아, 경로에 추가한다.
 3. 모든 도시가 경로에 포함될 때까지 2단계를 반복한다.
 4. 모든 도시를 방문한 후, 마지막 도시에서 시작 도시로 돌아가는 경로를 추가한다.

Lab: TSP를 A*로 해결

- 다음과 같은 6개의 도시와 그들간의 거리가 인접 행렬로 주어졌다고 가정하자.



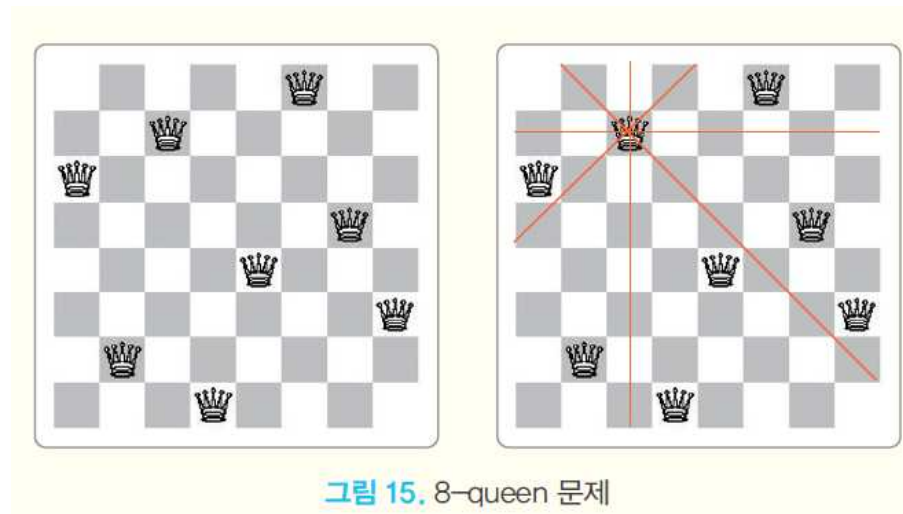
| | | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 5 | 2 | 6 | 4 | 5 |
| 1 | 5 | 0 | 7 | 1 | 3 | 6 |
| 2 | 2 | 7 | 0 | 6 | 3 | 3 |
| 3 | 6 | 1 | 6 | 0 | 2 | 4 |
| 4 | 4 | 3 | 3 | 2 | 0 | 5 |
| 5 | 5 | 6 | 3 | 4 | 5 | 0 |

| | | | | | | |
|---|----|---|----|---|---|---|
| | 0 | 1 | 2* | 3 | 4 | 5 |
| 0 | 0 | 5 | 2* | 6 | 4 | 5 |
| 1 | 5 | 0 | 7 | 1 | 3 | 6 |
| 2 | 2* | 7 | 0 | 6 | 3 | 3 |
| 3 | 6 | 1 | 6 | 0 | 2 | 4 |
| 4 | 4 | 3 | 3 | 2 | 0 | 5 |
| 5 | 5 | 6 | 3 | 4 | 5 | 0 |

- $0 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 0$ 순서로 도시를 방문합니다.

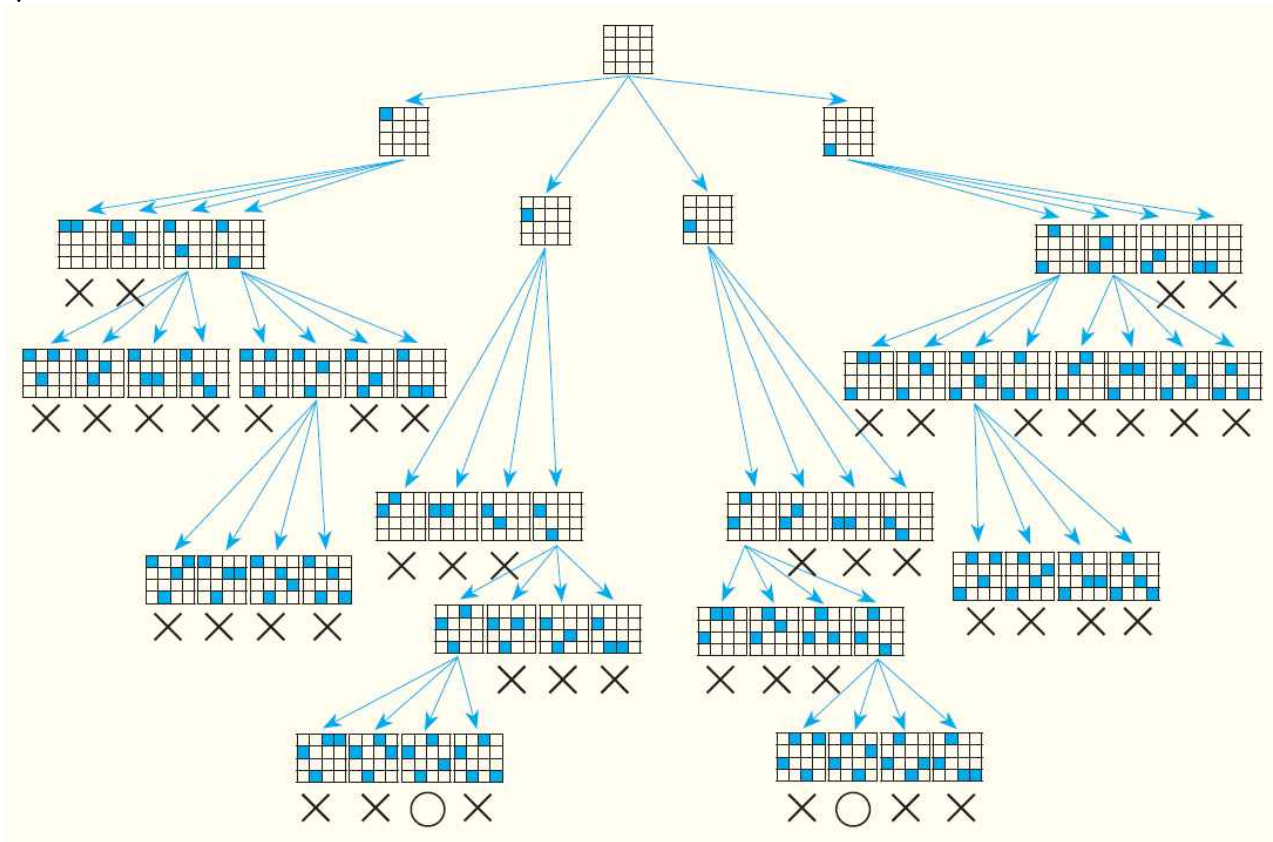
Lab: N-queen 문제의 탐색 알고리즘

- N-queen 문제는 $N \times N$ 크기의 체스판에 N개의 퀸을 서로 공격할 수 없도록 올려놓는 문제이다



Lab: N-queen 문제의 탐색 알고리즘

- 일단 문제를 간단하게 하기 위하여 하나의 퀸이 하나의 열에서만 움직인다고 가정하자. 하나의 열을 하나의 퀸이 장악하고 있다고 생각하자.



Lab: N-queen 문제의 탐색 알고리즘

- N-queen 문제에서 사용할 수 있는 휴리스틱은 어떤 것들이 있을까?

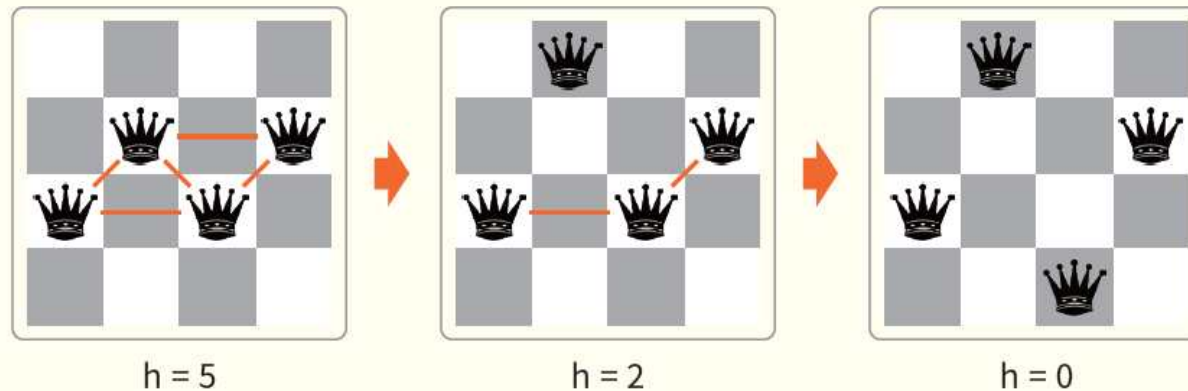


그림 16. 8-퍼즐에서의 휴리스틱 함수

실습

- N-queen 문제를 A* 탐색 방법을 사용하여 파이썬으로 구현하라.
 - 정수 N을 입력받아 N개의 queen이 서로 충돌하지 않도록 N*N 보드에 배치하는 문제를 A* 탐색 방법을 사용하여 구현

Summary

- 탐색은 상태 공간에서 시작 상태에서 목표 상태까지의 경로를 찾는 것이다. 연산자는 하나의 상태를 다른 상태로 변경한다.
- 맹목적인 탐색 방법(**blind search method**)은 목표 노드에 대한 정보를 이용하지 않고 기계적인 순서로 노드를 확장하는 방법이다. 깊이 우선 탐색과 너비 우선 탐색, 반복적 깊이 증가 탐색이 있다.
- 탐색에서는 중복된 상태를 막기 위하여 **OPEN** 리스트와 **CLOSED** 리스트를 사용한다.
- 경험적 탐색 방법(**heuristic search method**)은 목표 노드에 대한 경험적인 정보를 사용하는 방법이다. “언덕 등반 기법” 탐색, 최상 우선 탐색과 **A*** 탐색이 있다.
- **A*** 알고리즘은 $f(n) = g(n) + h(n)$ 으로 생각한다. $h(n)$: 현재 노드에서 목표 노드까지의 거리이고, $g(n)$: 시작 노드에서 현재 노드까지의 비용이다.

Q & A

