제2장 탐색

이번 장에서 다루는 내용

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

알파고는 어떻게 수를 읽었을까?

- 알파고는 딥러닝과 *탐색 기법을* 통하여 다음 수를 읽었다.

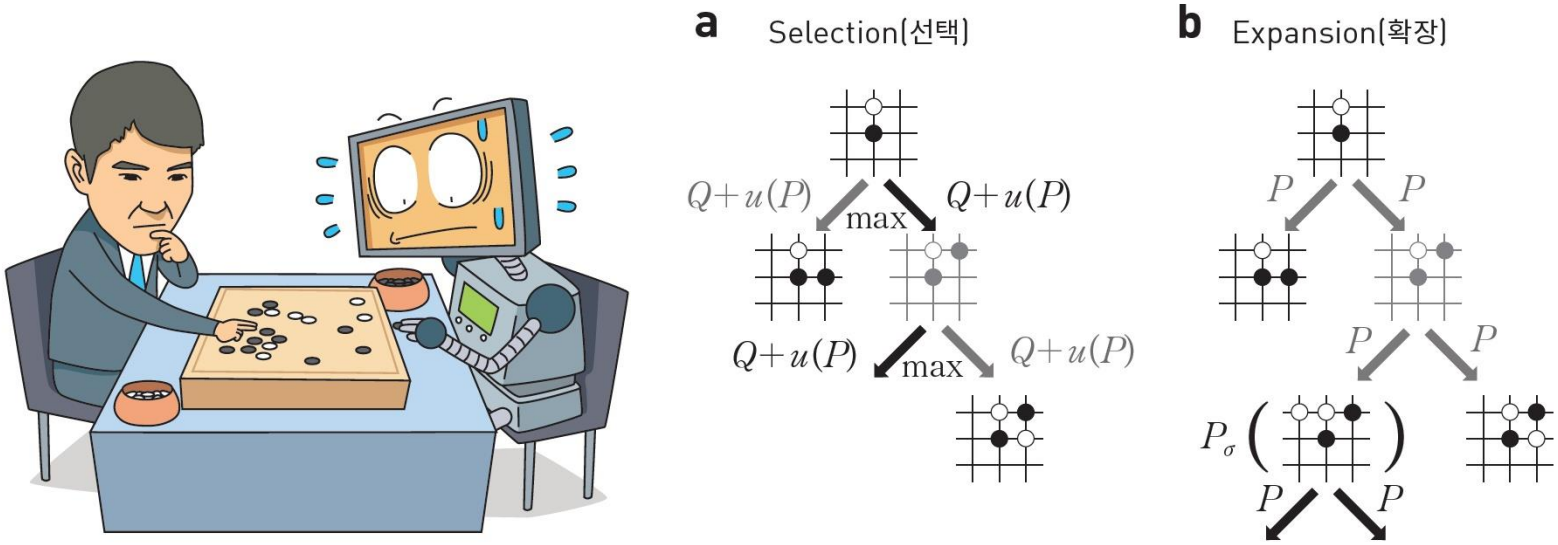


그림 2-1 알파고에서의 몬테카를로 트리 탐색(오른쪽 그림 출처: 알파고 네이처 논문)

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

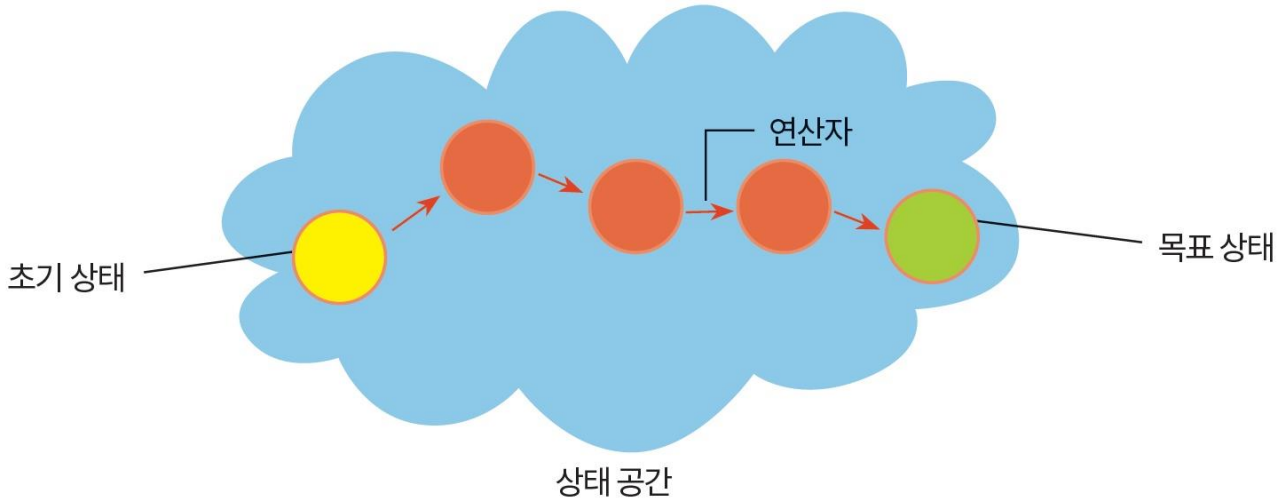
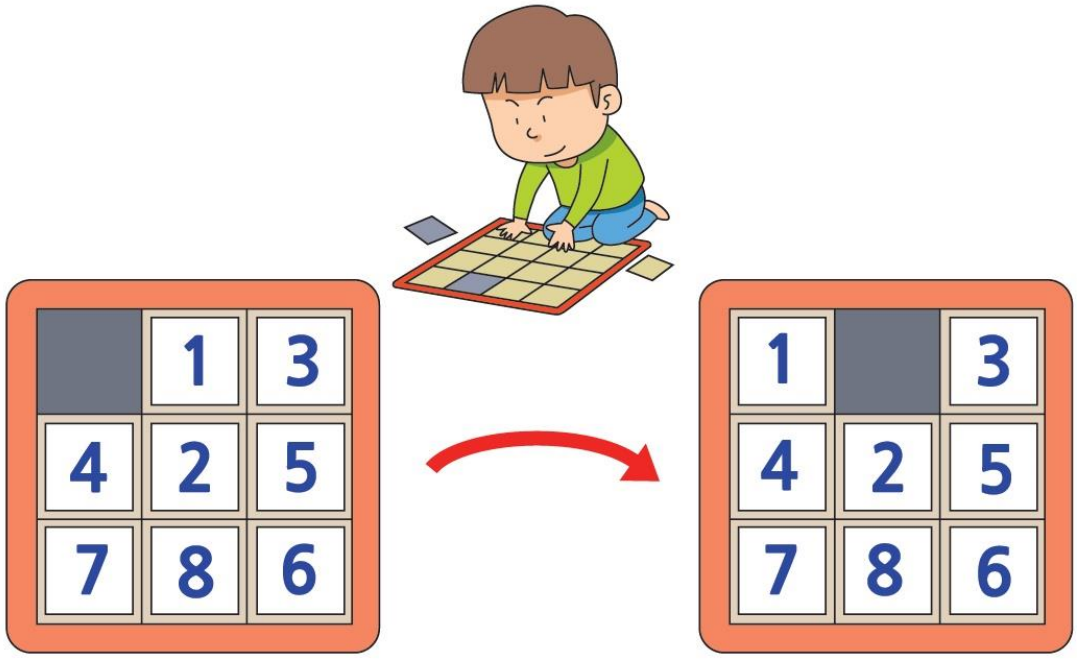


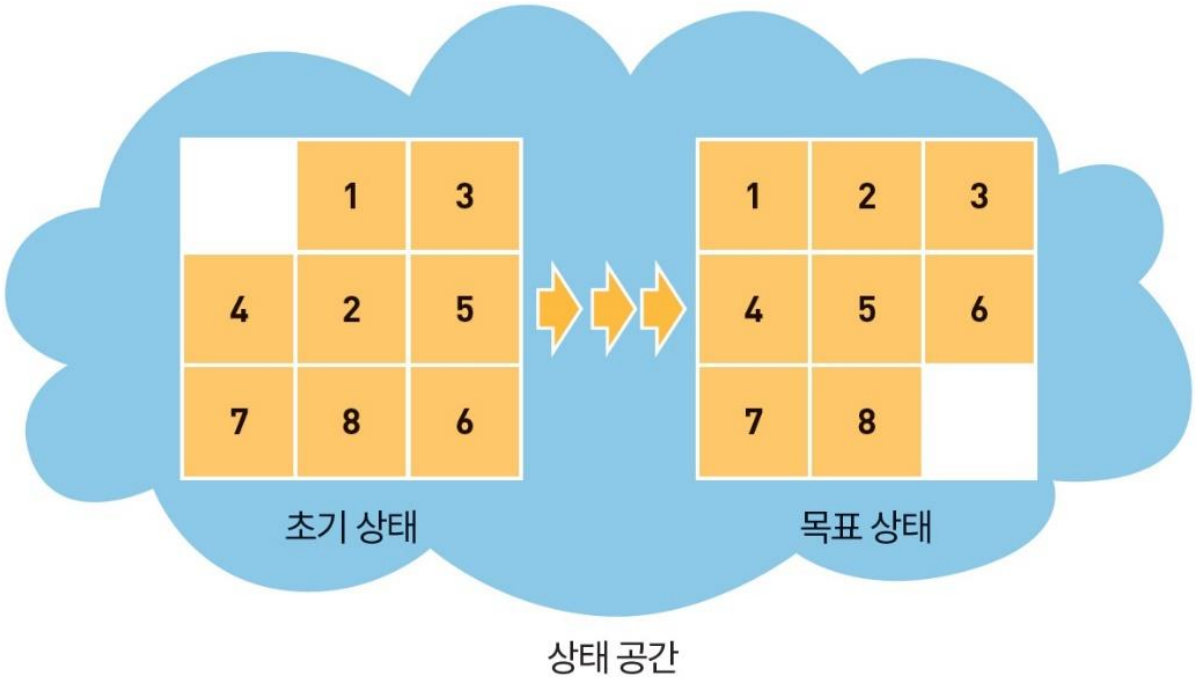
그림 2–2 상태, 상태 공간, 연산자

8-puzzle

- http://www.puzzlopia.com/puzzles/puzzle-8/play



8-puzzle



8-puzzle에서의 연산자

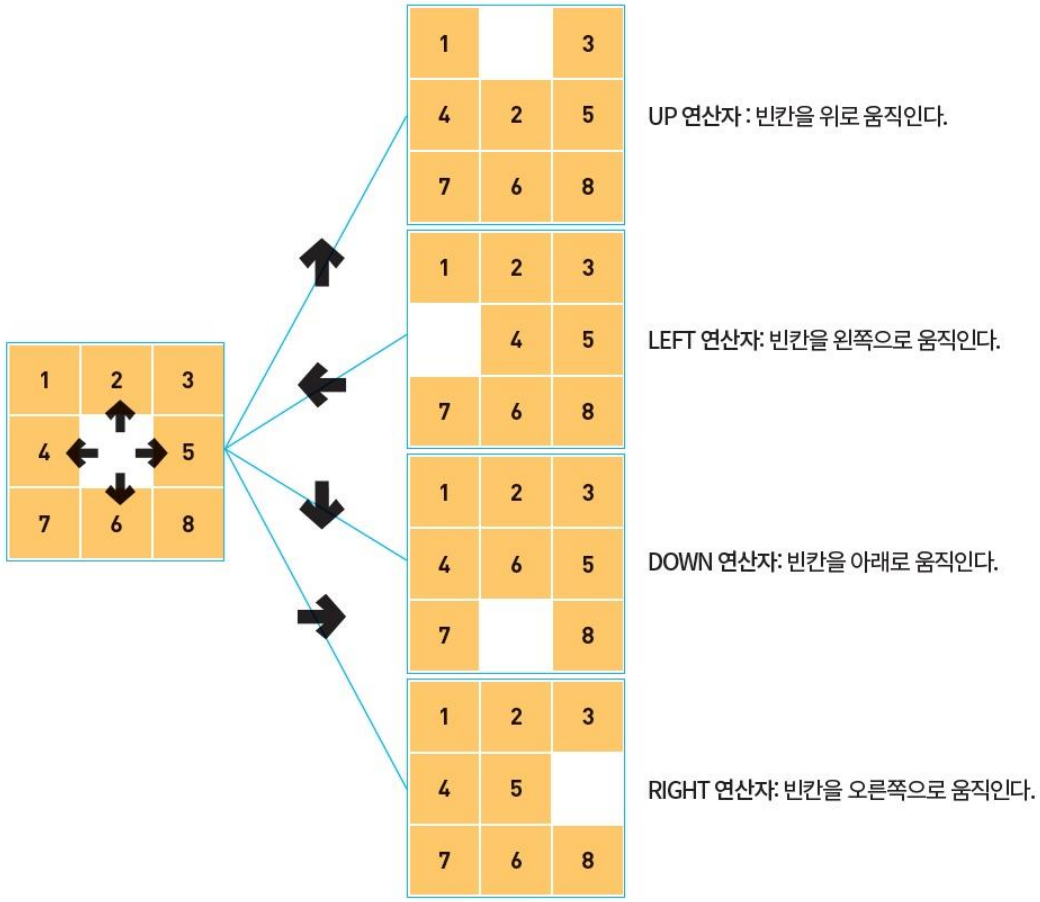


그림 2-4 상하좌우 연산자

8-puzzle 에서의 상태 공간

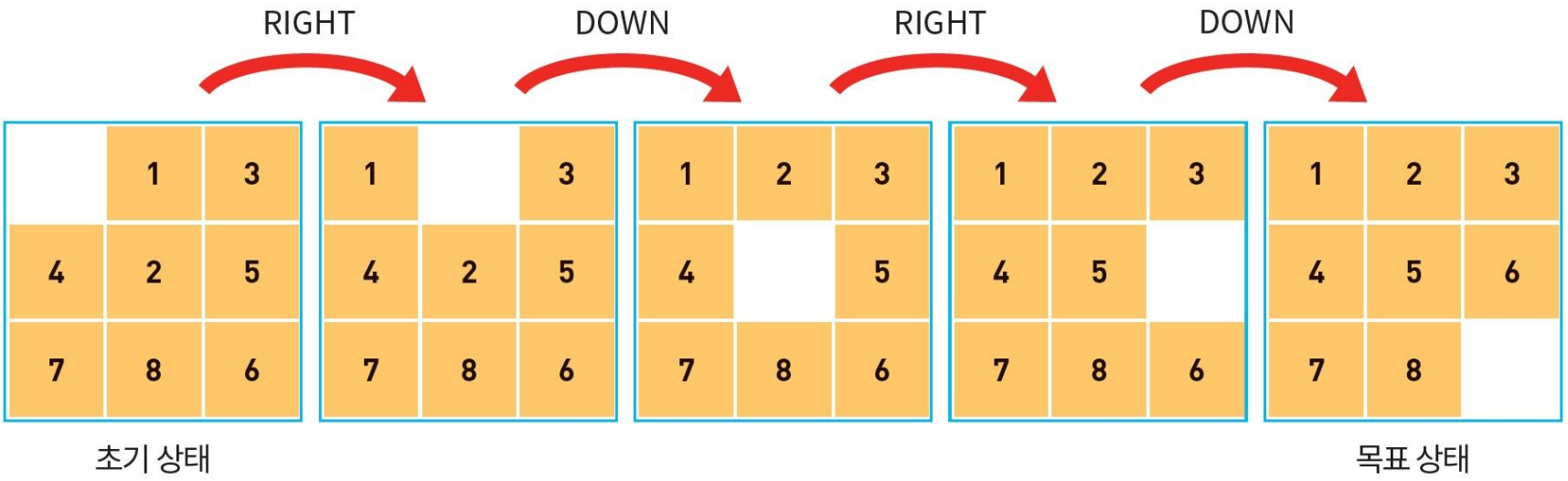


그림 2–5 연산자를 사용하여 이동하는 8–퍼즐 예제

Lab: 하노이 탑

- 상태?
- 연산자?

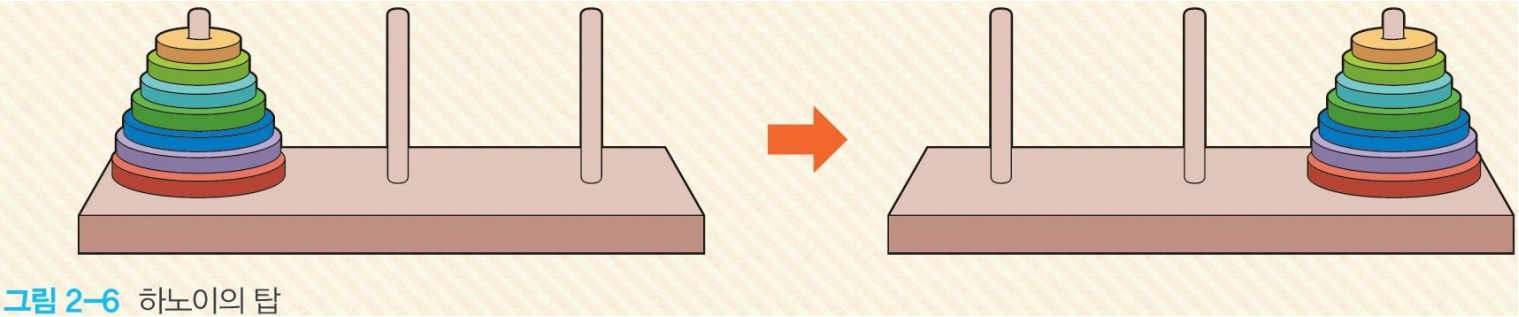


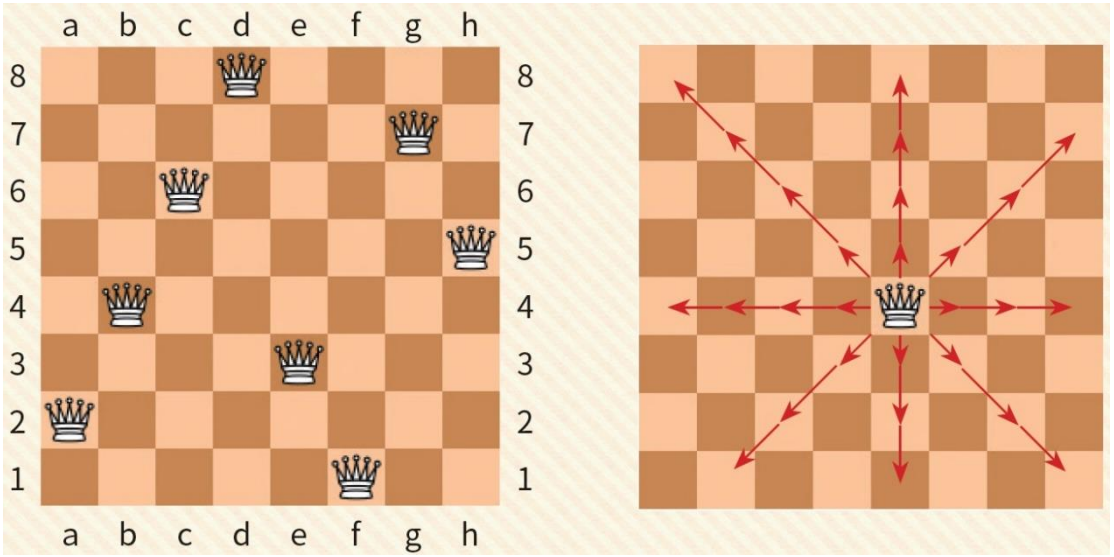
그림 2–6 하노이의 탑

Lab: 하노이 탑

- 상태공간 A = { (a_1,a_2,a_3) I $a_i \in \{A, B, C\}$ }
- 초기상태 I = (A, A, A)
- 목표 상태 G = (C, C, C)
- 연산자 O = {$move_{which,\ where}$ I which $\in$ {1,2,3}, where$\in${A, B, C}}

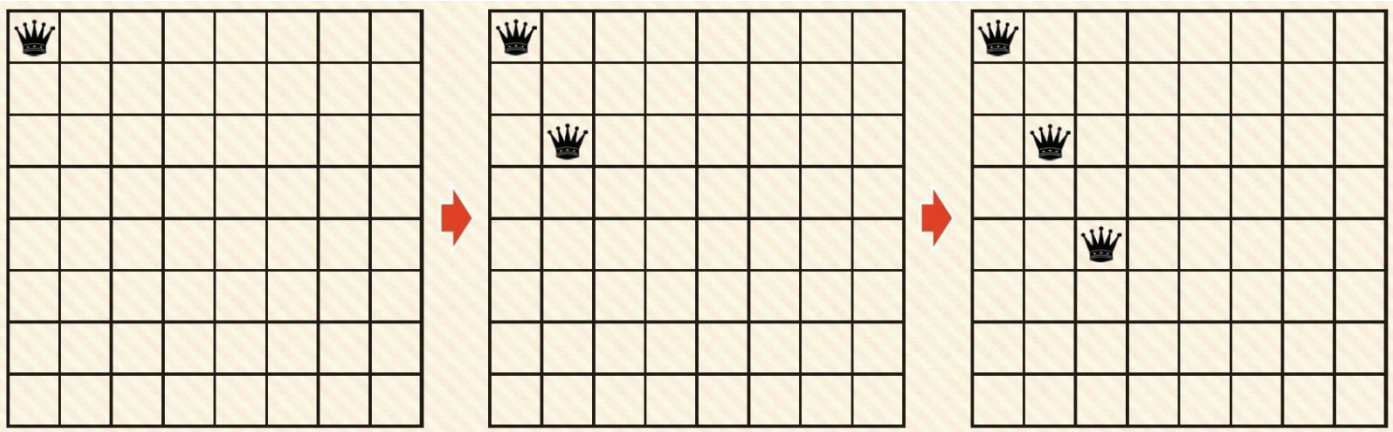
Lab:N-queen

- 상태?
- 연산자?



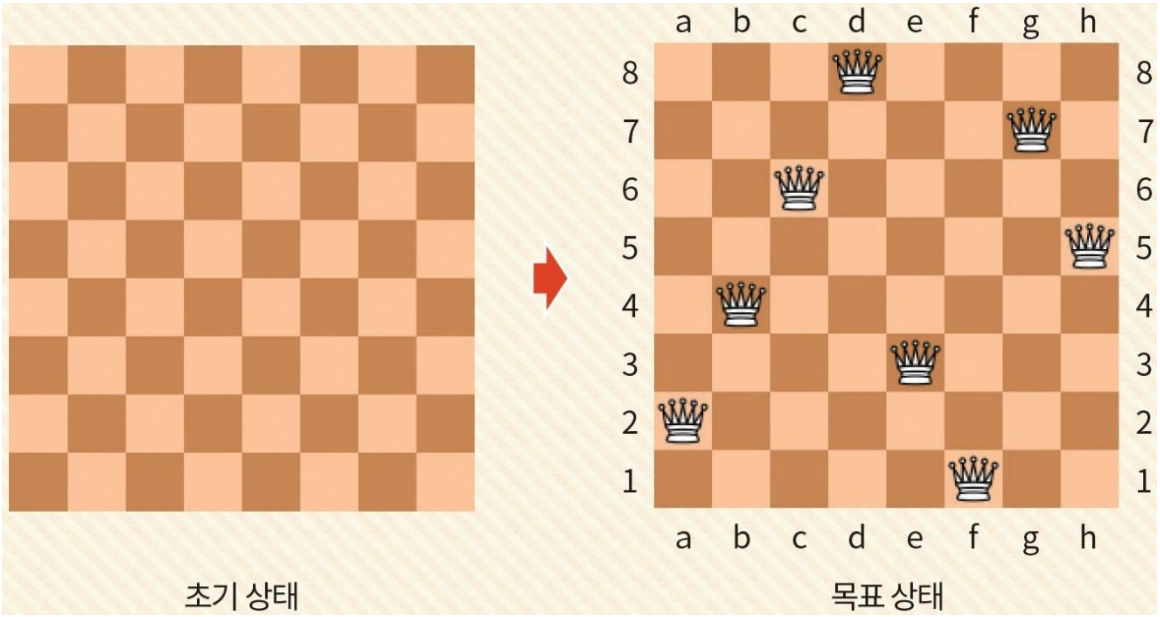
Lab:N-queen

- 각각의 퀸을 정해진 열에서만 움직이게 한다.



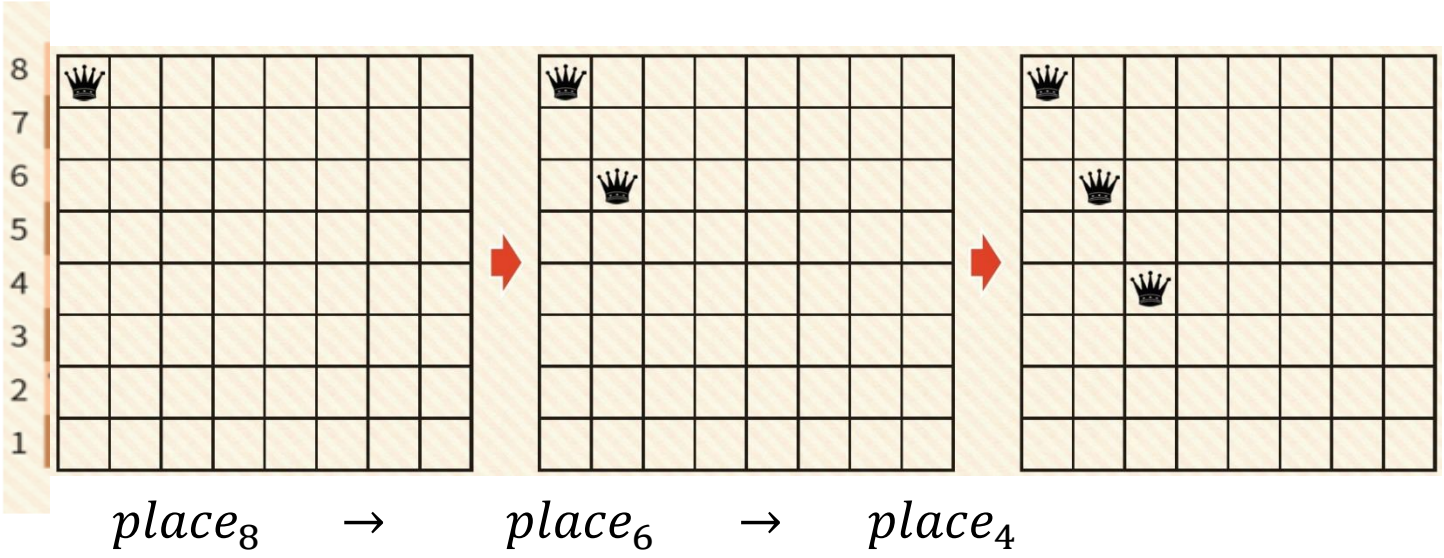
Lab:N-queen

- 초기 상태와 목표 상태



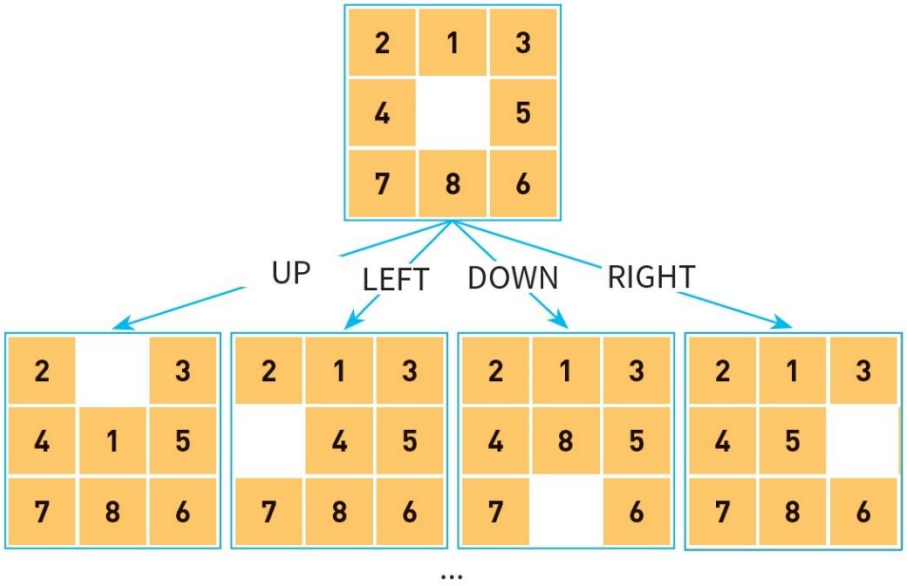
Lab:N-queen

- 연산자 집합 $O = \{ \text{place}_i \mid 1 \le i \le 8 \}$
- $place_i$연산자는 새로운 퀸을 i번째 행에 배치한다.



탐색 트리

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



탐색 트리

- 연산자를 적용하기 전까지는 탐색 트리는 미리 만들어져 있지 않음!-

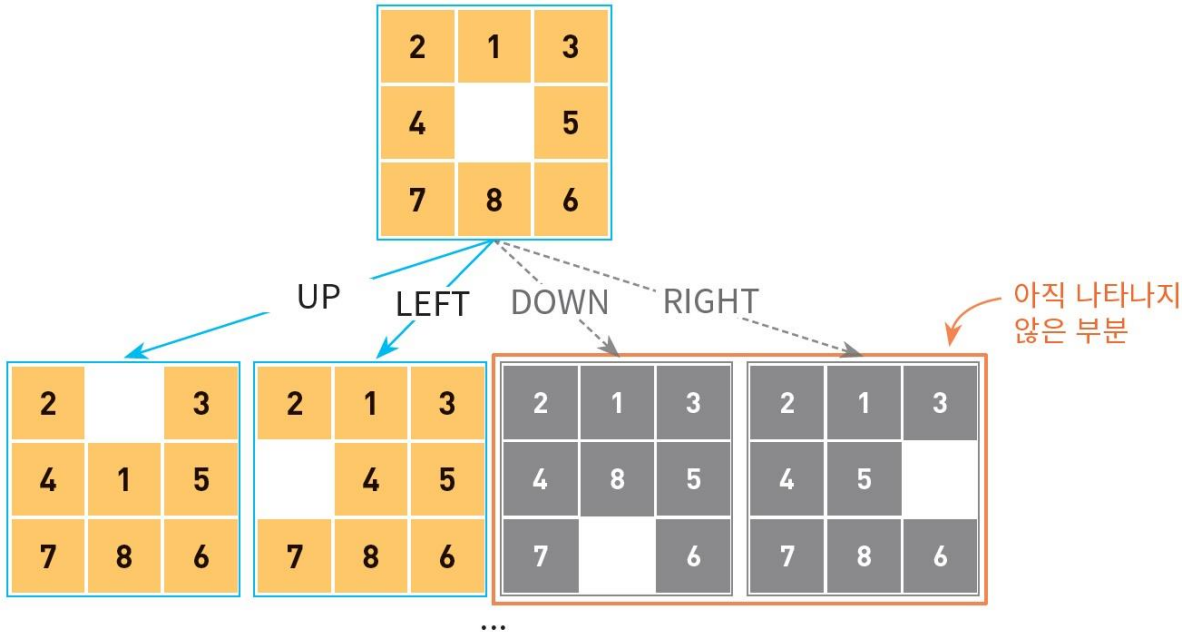
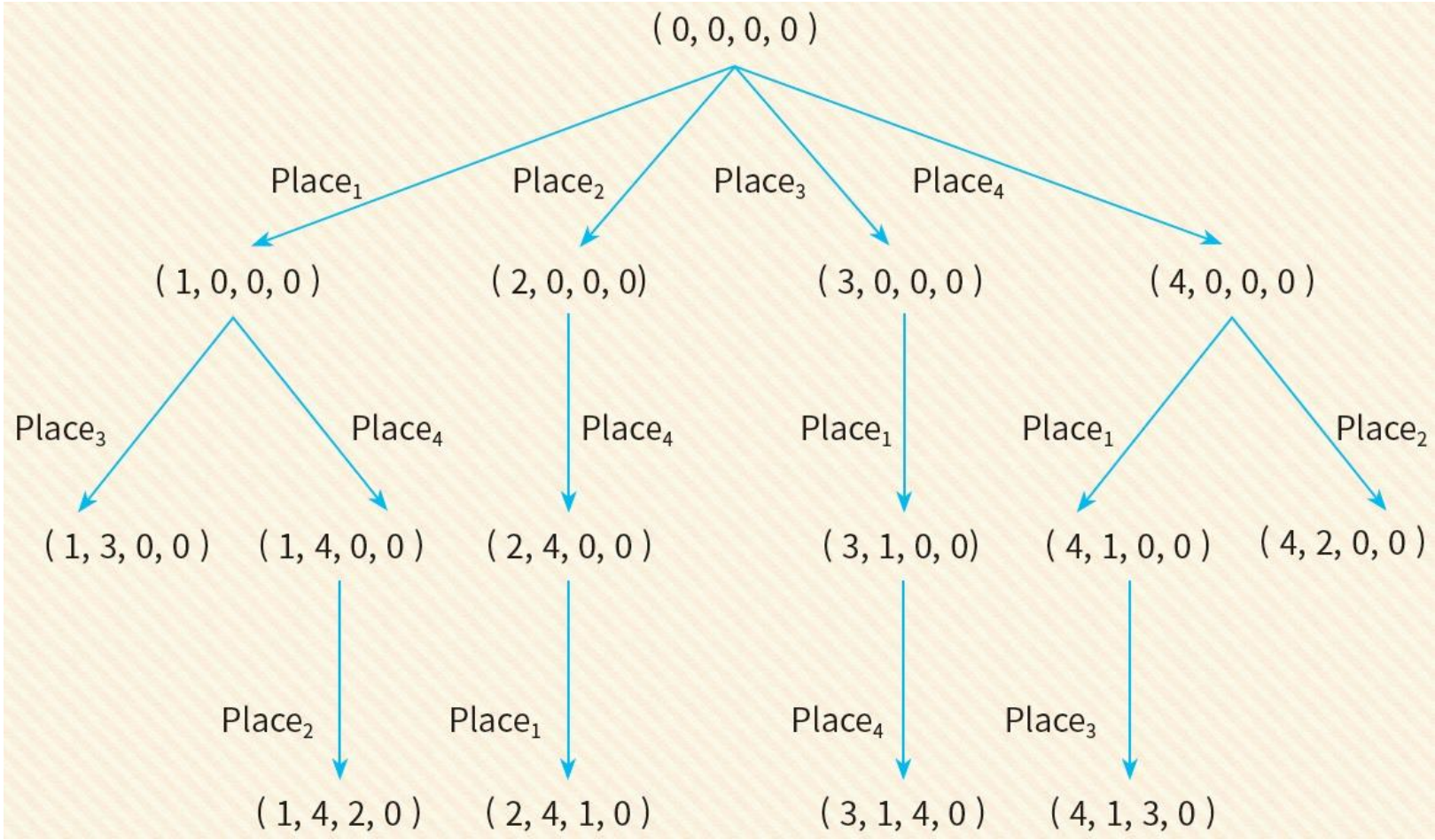


그림 2-9 탐색 트리의 노드는 동적으로 생성된다.

Lab: 4-queen 문제 탐색 트리의 일부



기본적인 탐색 기법

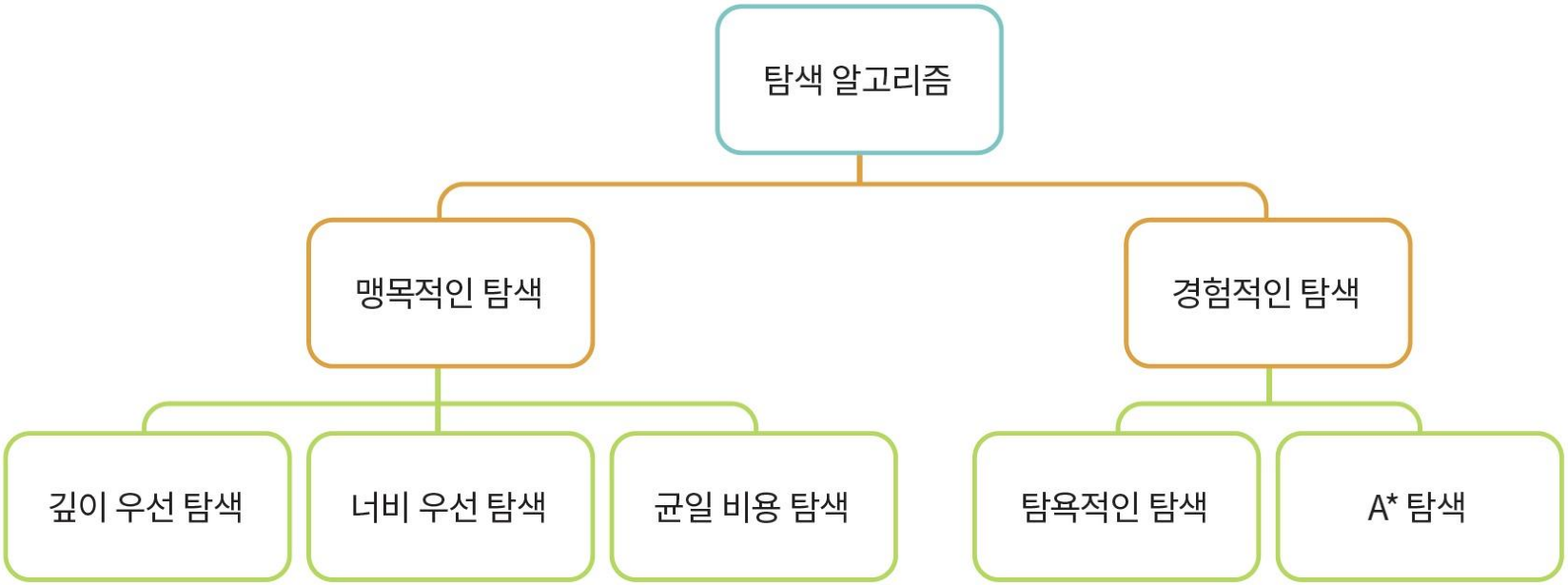


그림 2-10 탐색의 종류

깊이 우선 탐색(DFS)

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

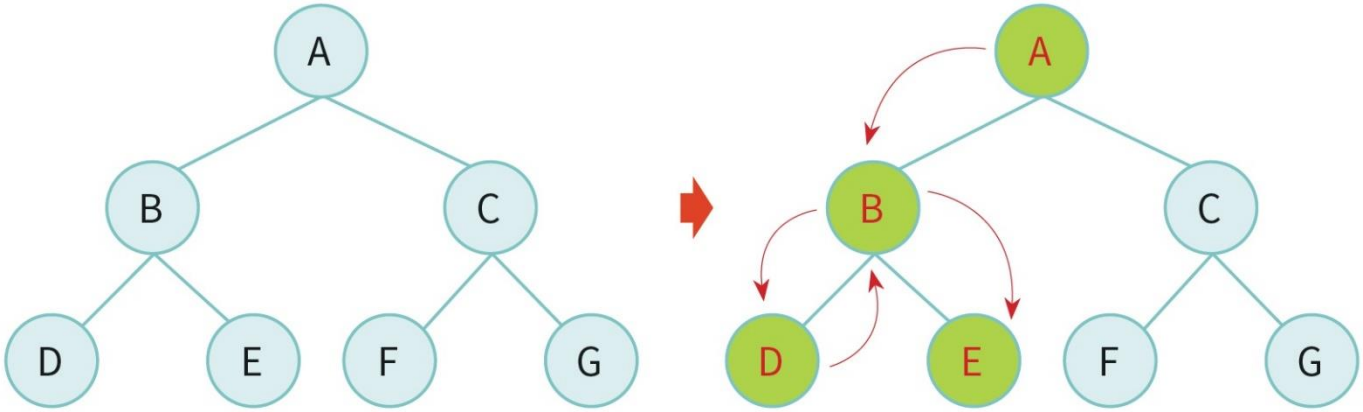


그림 2-11 깊이 우선 탐색

깊이 우선 탐색(8-puzzle)

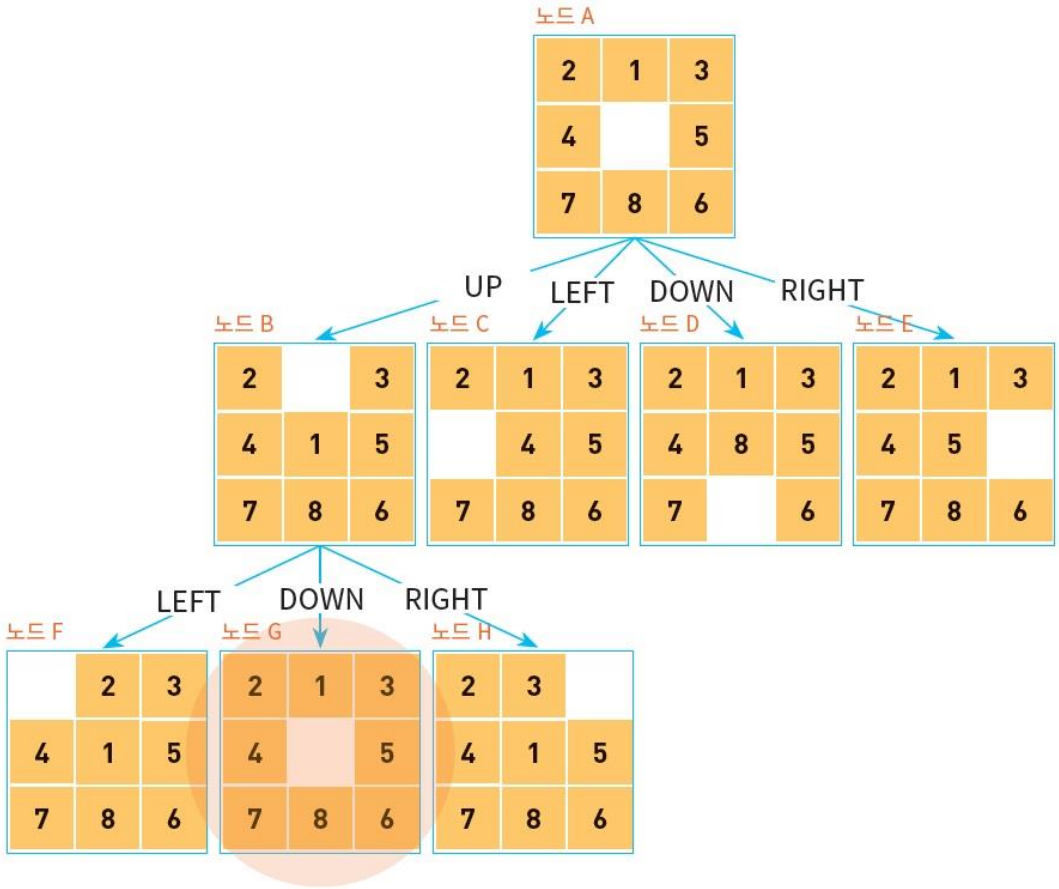
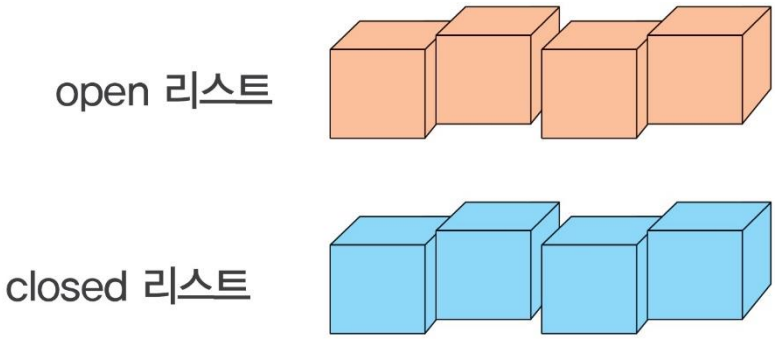


그림 2-12 8-퍼즐에서의 깊이 우선 탐색

OPEN CLOSED 리스트

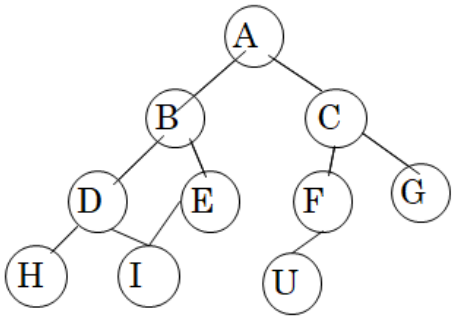
- 탐색에서는 중복된 상태를 막기 위하여 다음과 같은 2개의 리스트를 사용한다.
  - OPEN 리스트: 확장은 되었으나 아직 탐색하지 않은 상태들이 들어 있는 리스트
  - CLOSED 리스트: 탐색이 끝난 상태들이 들어 있는 리스트



DFS 알고리즘

```
depth_first_search()
open ← [시작노드]
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




```
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!
```

너비 우선 탐색(BFS)

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

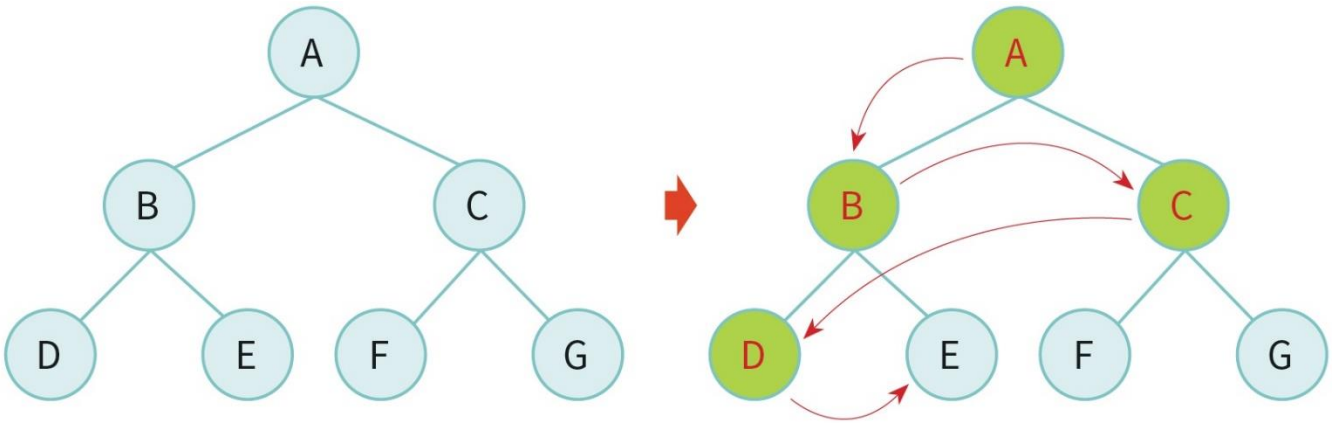
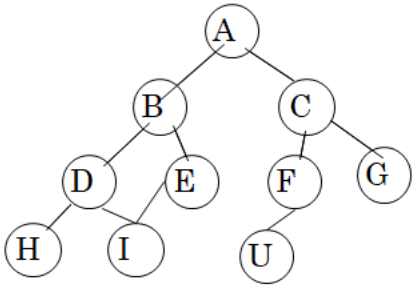


그림 2-13 너비 우선 탐색

BFS 알고리즘

```
depth_first_search()
open ← [시작노드]
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 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!

DFS와 BFS 프로그램

- 8-puzzle을 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
```

게임 보드 표현

```
class State:
  def __init__(self, board, goal, moves=0):
    self.board = board
    self.moves = moves
    self.goal = goal
...
```

상태 표현

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


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

OPEN과 CLOSED 리스트

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

# closed 리스트
closed_queue = [ ]
```

자식 노드 생성

```
# 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, 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))
   if not i in [2, 5, 8]:                     # DOWN 연산자
     result.append(self.get_new_board(i, i+1, moves))
   if not i in [6, 7, 8]:                     # RIGHT 연산자
     result.append(self.get_new_board(i, i+3, moves))
   return result
```

전체 소스 #1

```
# 상태를 나타내는 클래스
class State:
  def __init__(self, board, goal, moves=0):
    self.board = board
    self.moves = moves
    self.goal = goal

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

전체 소스 #2

```
# 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, 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))
    if not i in [2, 5, 8]:                     # DOWN 연산자
      result.append(self.get_new_board(i, i+1, moves))
    if not i in [6, 7, 8]:                     # RIGHT 연산자
      result.append(self.get_new_board(i, i+3, moves))
    return result
```

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"+\
    "-----------------"

# 초기 상태
puzzle = [1, 2, 3,
          0, 4, 6,
          7, 5, 8]

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

전체 소스 #4

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

closed_queue = [ ]
moves = 0
while len(open_queue) != 0:
  current = open_queue.pop(0)                    # OPEN 리스트의 앞에서 삭제
  print(current)
  if current.board == goal:
      print("탐색 성공")
      break
  moves = current.moves+1
  closed_queue.append(current)
  for state in current.expand(moves):
      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, 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
```

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.

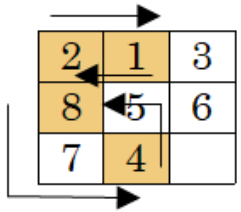
2	1	3
8	5	6
7	4	

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



언덕 등반 기법

- 이 기법에서는 평가 함수의 값이 좋은 노드를 먼저 처리한다.
- 평가함수로 제 위치에 있지 않은 타일의 개수 사용

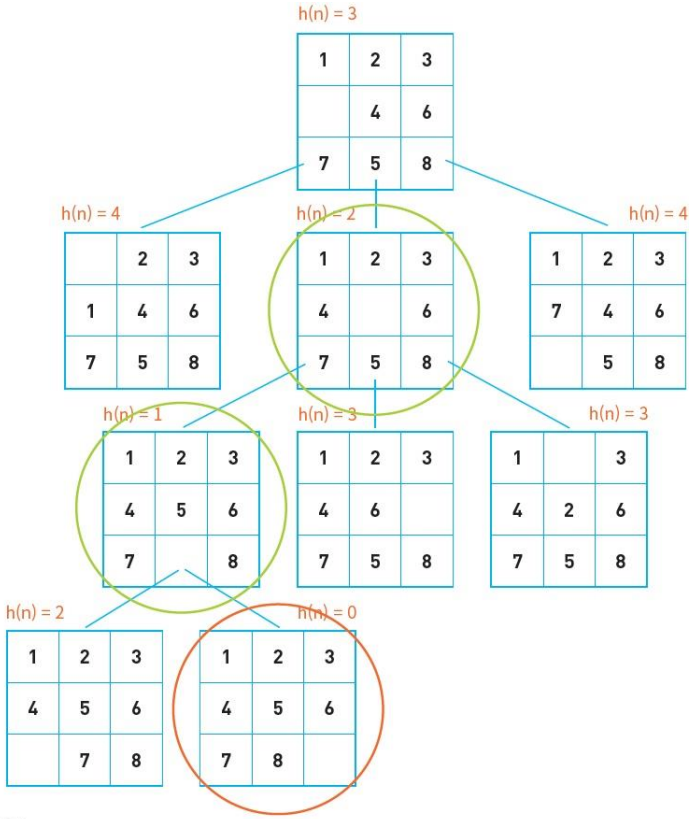
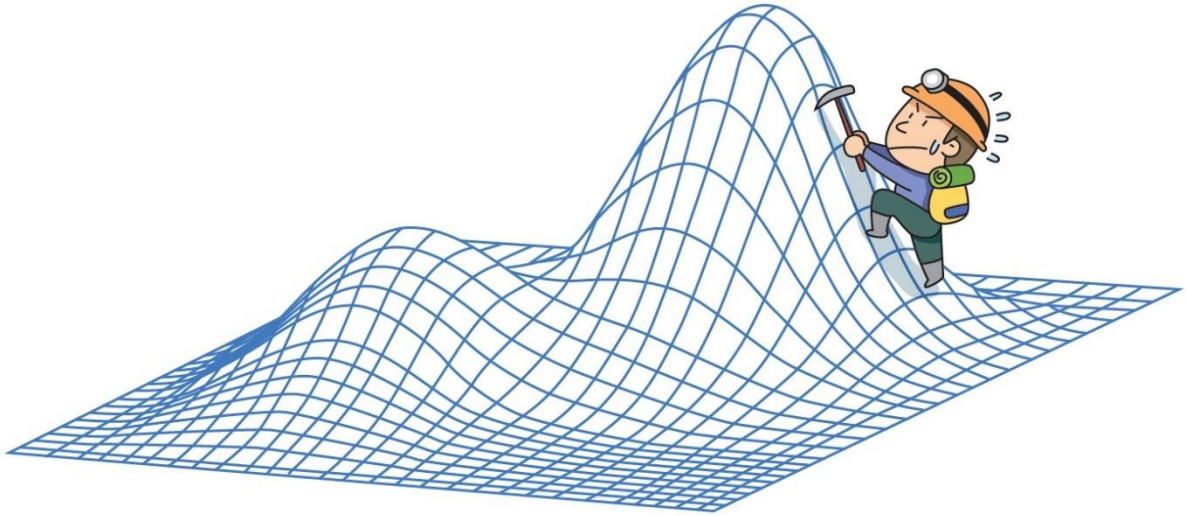


그림 2-14 언덕 등반 기법

언덕 등반 기법

- 경험적인 탐색 방법은 무조건 휴리스틱 함수 값이 가장 좋은 노드만을 선택한다.
- 이것은 등산할 때 무조건 현재의 위치보다 높은 위치로만 이동하는 것과 같다. 일반적으로는 현재의 위치보다 높은 위치로 이동하면 산의 정상에 도달할 수 있다.

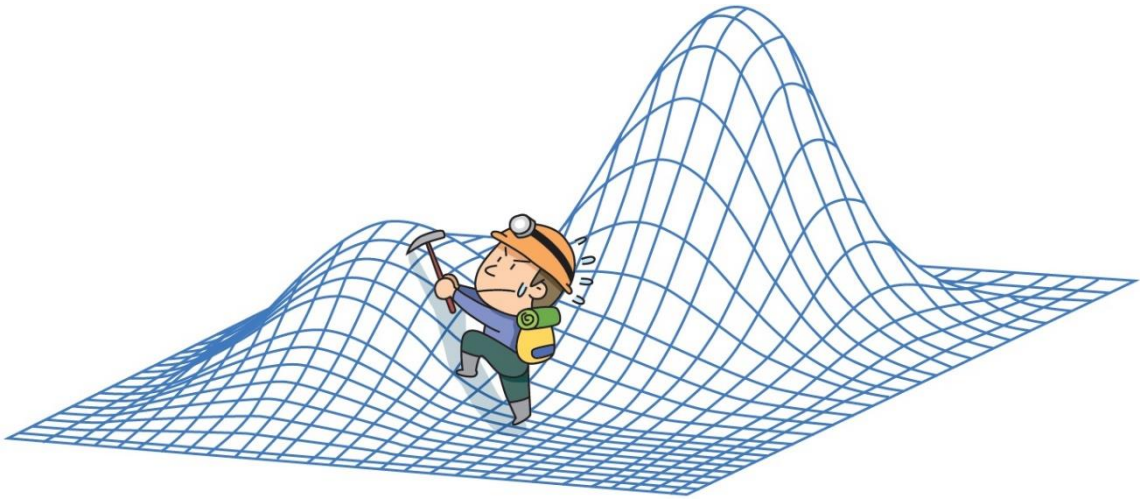


언덕 등반 기법 알고리즘

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).

지역 최소 문제

- 순수한 언덕 등반 기법은 오직 h(n) 값만을 사용한다(OPEN 리스트나 CLOSED 리스트도 사용하지 않는다).
- 이런 경우에는 생성된 자식 노드의 평가함수 값이 부모 노드보다 더 높거나 같은 경우가 나올 수 있다. 이것을 지역 최소 문제(local minima problem)라고 한다.



지역 최소 문제의 예

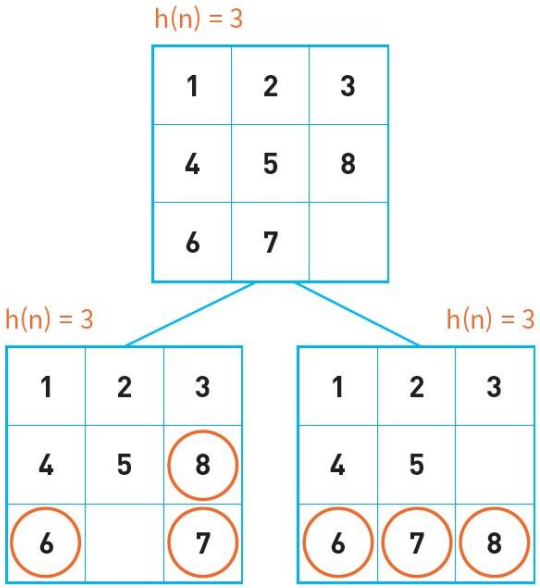


그림 2–15 지역 최소 문제

A* 알고리즘

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

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

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

8-puzzle 에서의 A* 알고리즘

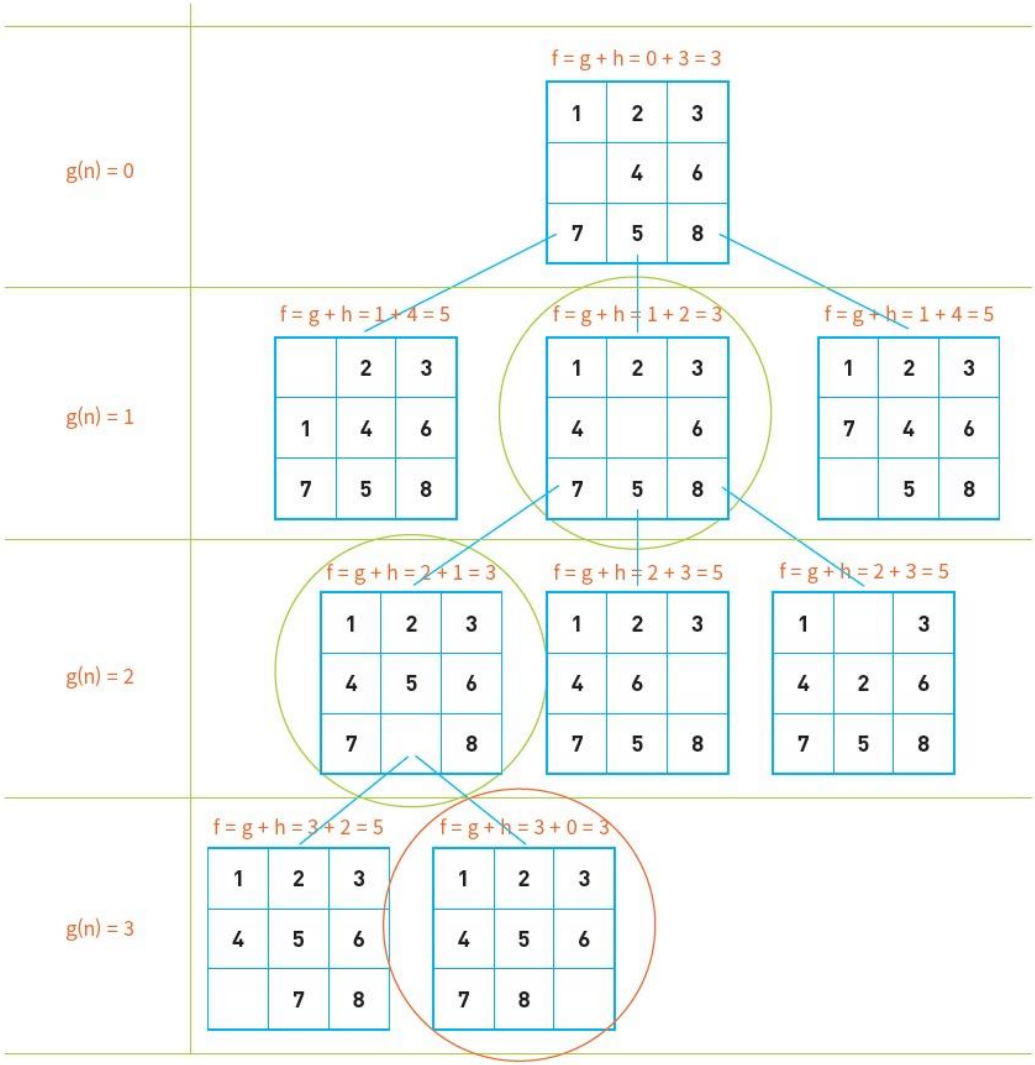


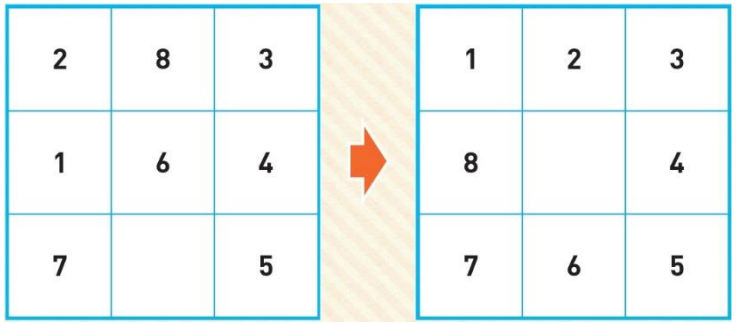
그림 2-17 8-퍼즐에서의 A* 알고리즘

A* 알고리즘

```
AStar_search()

open ← [시작노드]
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 open이나 closed에 있지 않으면
                        자식 노드의 평가 함수 값 f(n) = g(n) + h(n)을 계산한다.
                        자식 노드들을 open에 추가한다.
return FAIL
```

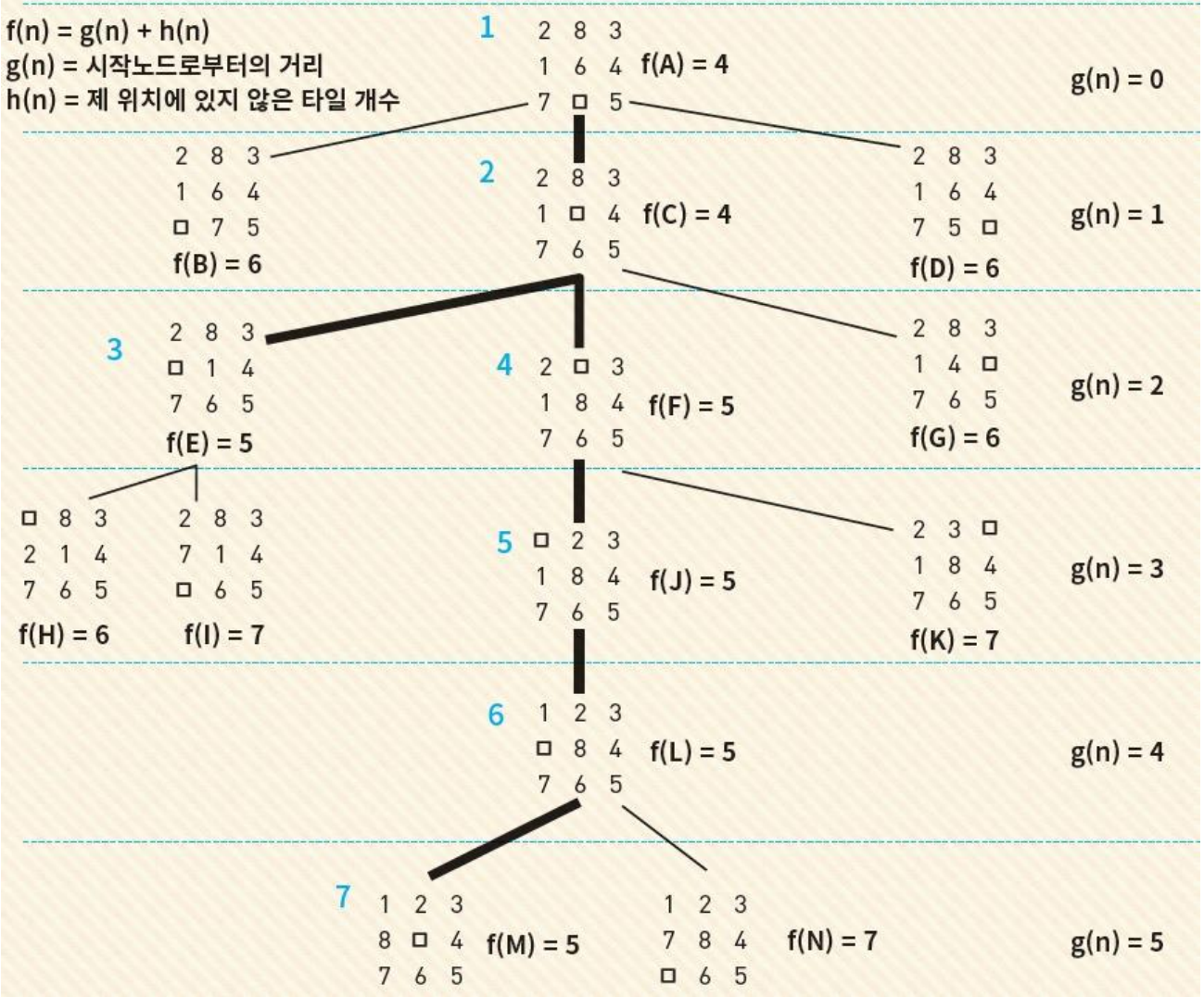
Lab: A* 알고리즘을 시뮬레이션

- 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 파이썬 구현

```
import queue

# 상태를 나타내는 클래스, f(n) 값을 저장한다.
class State:
  def __init__(self, board, goal, moves=0):
    self.board = board
    self.moves = moves
    self.goal = goal

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

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, 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))
   if not i in [2, 5, 8]:                     # DOWN 연산자
     result.append(self.get_new_board(i, i+1, moves))
   if not i in [6, 7, 8]:                     # RIGHT 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):
   return sum([1 if self.board[i] != self.goal[i] else 0 for i in range(8)])

 # 시작 노드로부터의 경로를 반환한다.
 def g(self):
   return self.moves

 # 상태와 상태를 비교하기 위하여 less than 연산자를 정의한다.
 def __lt__(self, other):
   return self.f() < other.f()
```

A* 알고리즘 파이썬 구현

```
 # 객체를 출력할 때 사용한다.
  def __str__(self):
    return "----------------- f(n)=" + str(self.f()) +"\n"+\
    "----------------- h(n)=" + str(self.h()) +"\n"+\
    "----------------- g(n)=" + str(self.g()) +"\n"+\
    str(self.board[:3]) +"\n"+\
    str(self.board[3:6]) +"\n"+\
    str(self.board[6:]) +"\n"+\
    "-----------------"

# 초기 상태
puzzle = [1, 2, 3,
          0, 4, 6,
          7, 5, 8]
# 목표 상태
goal = [1, 2, 3,
        4, 5, 6,
        7, 8, 0]

# open 리스트는 우선순위 큐로 생성한다.
open_queue = queue.PriorityQueue()
open_queue.put(State(puzzle, goal))
```

A* 알고리즘 파이썬 구현

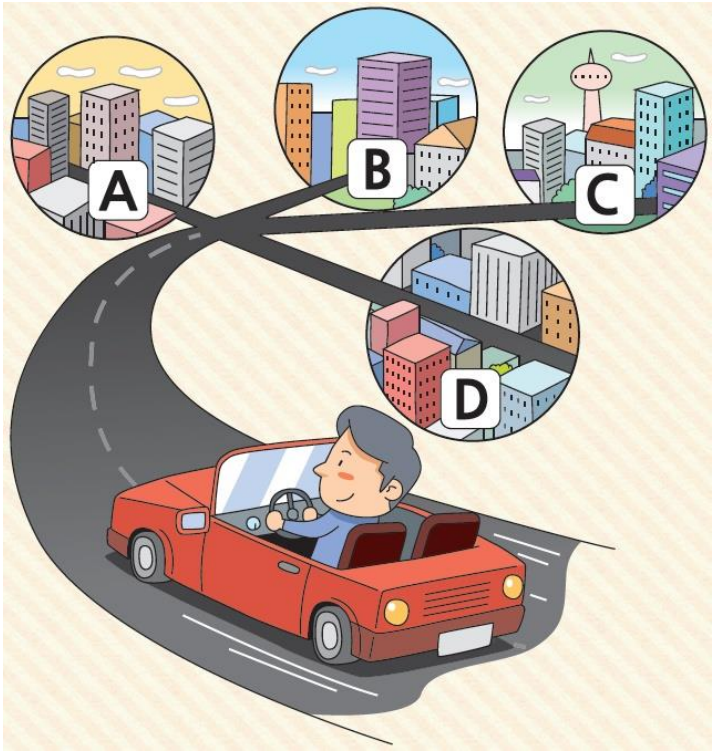
```
closed_queue = [ ]
moves = 0
while not open_queue.empty():

  current = open_queue.get()
  print(current)
  if current.board == goal:
      print("탐색 성공")
      break
  moves = current.moves+1
  for state in current.expand(moves):
    if state not in closed_queue:
      open_queue.put(state)
  closed_queue.append(current)
else:
  print ('탐색 실패')
```

실행 결과

```
---------- f(n)= 3
---------- h(n)= 3
---------- g(n)= 0
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
----------
---------- f(n)= 3
---------- h(n)= 2
---------- g(n)= 1
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
----------
...
---------- f(n)= 3
---------- h(n)= 0
---------- g(n)= 3
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
----------
탐색 성공
```

Lab: TSP

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



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 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" 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