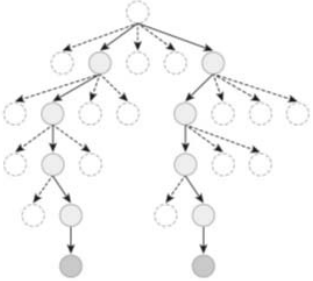
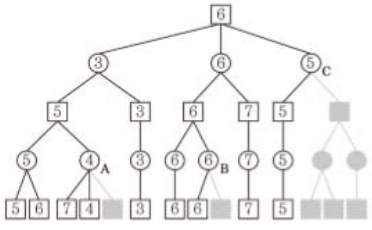
제2장 탐색

인공 지능의 연구 분야 – 요소 기술

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





인공 지능의 요소 기술

- 지식표현(knowledge representation)
  - **문제 해결**에 **이용**하거나 **심층적 추론**을 할 수 있도록 **지식**을 효과적으로 **표현**하는 방법
  - **IF-THEN 규칙**(rule)
  - **프레임**(frame): 관련된 정보를 slot과 daemon procedure로 구성
  - **의미망**(semantic net): 네트워크의 형태로 관련 지식을 표현
  - **논리**(logic) : 명제논리(propositional logic), 술어논리(predicate logic)
  - **스크립트**: 절차적 지식을 표현
  - **퍼지 논리**(fuzzy logic): 애매한 지식을 표현
  - **확률적 방법**: 불확실한 지식을 표현
  - **확률 그래프 모델**
  - **온톨로지 기술 언어 : RDF, OWL**

인공 지능의 요소 기술

- 추론(inference)
  - **가정**이나 **전제**로부터 **결론**을 이끌어내는 것
  - **규칙기반 시스템의 추론**
    - 전향추론(forward inference)
    - 후향추론(backward inference)
  - **확률 모델의 추론**
    - 관심 대상의 **확률** 또는 **확률분포**를 결정하는 것
    - 베이즈 정리(Bayesian theorem) 및 주변화(marginalization) 이용

$$P(A) = \sum_b P(A, B = b)$$

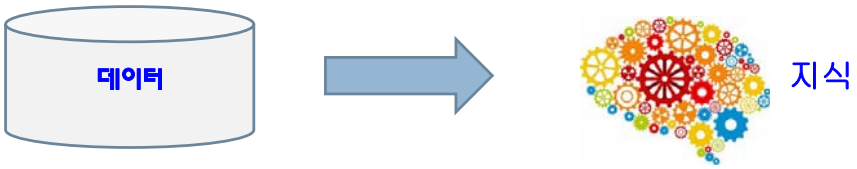
사후확률 가능도 사전확률

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

증거

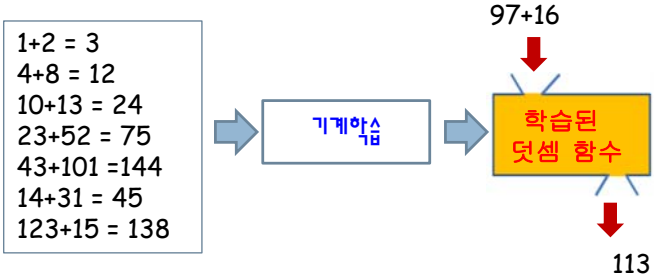
인공 지능의 요소 기술

- 기계 학습(machine learning)
  - 경험을 통해서 나중에 유사하거나 같은 일(task)를 더 효율적으로 처리할 수 있도록 **시스템**의 **구조**나 **파라미터**를 **바꾸는 것**
  - 컴퓨터가 **지식**을 갖게 만드는 작업
  - **지도학습 (supervised learning)**
    - 입력과 대응하는 출력을 데이터로 제공하고 대응관계의 함수 찾기
  - **비지도학습 (unsupervised learning)**
    - 데이터만 주어진 상태에서 유사한 것들을 서로 묶어 군집을 찾거나 확률분포 표현
  - **강화학습 (reinforcement learning)**
    - 상황 별 행동에 따른 시스템의 보상 값(reward value)만을 이용하여, 시스템에 대한 바람직한 행동 정책(policy) 찾기



인공 지능의 요소 기술

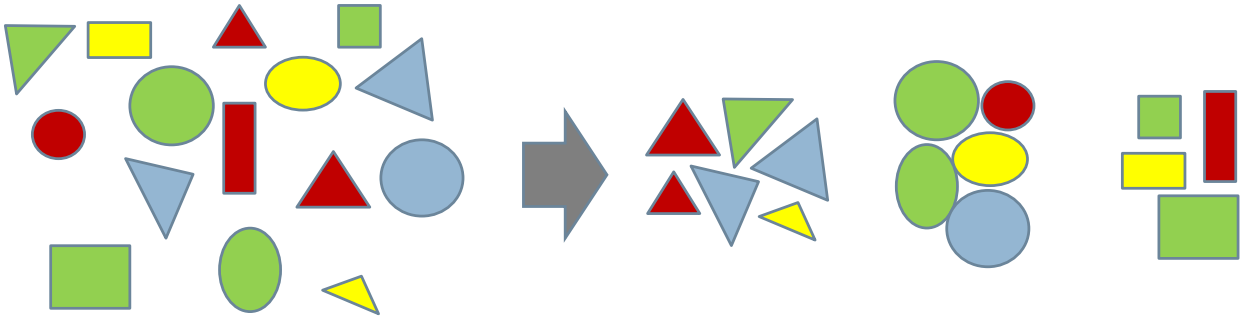
- 기계 학습(machine learning)
  - 지도학습(supervised learning)
    - **입력(문제)**과 대응하는 **출력(답)**을 데이터로 제공하고 대응관계의 **함수** 또는 **패턴**을 찾는 것



  - 분류(classification), 회귀(regression)

인공 지능의 요소 기술

- 기계 학습(machine learning) - cont.
  - 비지도학습(unsupervised learning)
    - 답이 없는 **문제들만 있는 데이터**들로부터 **패턴**을 추출하는 것



    - 군집화(clustering), 밀도추정(density estimation), 토픽 모델링

인공 지능의 요소 기술

- 기계 학습(machine learning) - cont.
  - 강화학습(reinforcement learning)
    - 문제에 대한 직접적인 답을 주지는 않지만 경험을 통해 **기대 보상**(expected reward)이 최대가 되는 **정책**(policy)을 찾는 학습

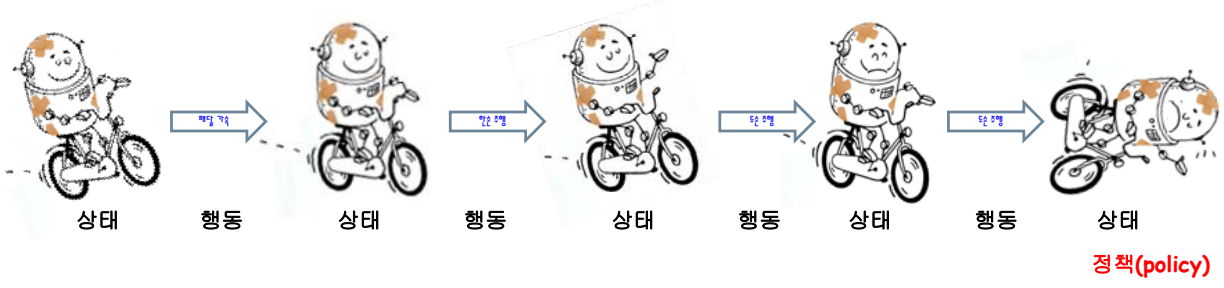


Image source : RÉMI MUNOS
RAPPORT ANNUEL INRIA 2008

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.

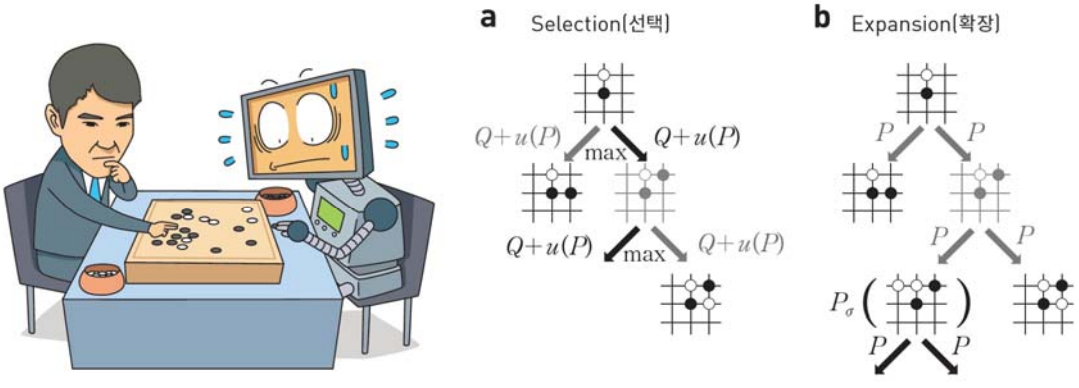


그림 2-1 알파고에서의 몬테카를로 트리 탐색(오른쪽 그림 출처: 알파고 네이처 논문)

c Play out(평가)

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

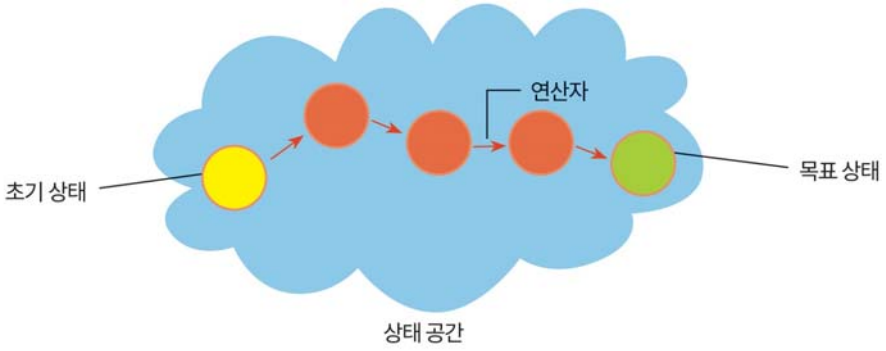
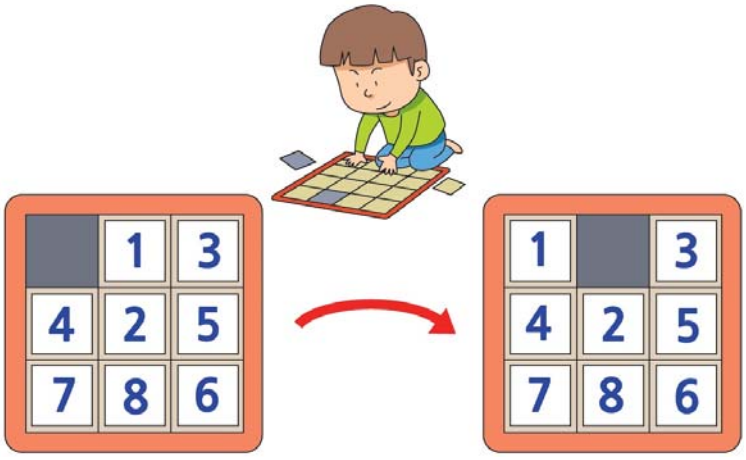
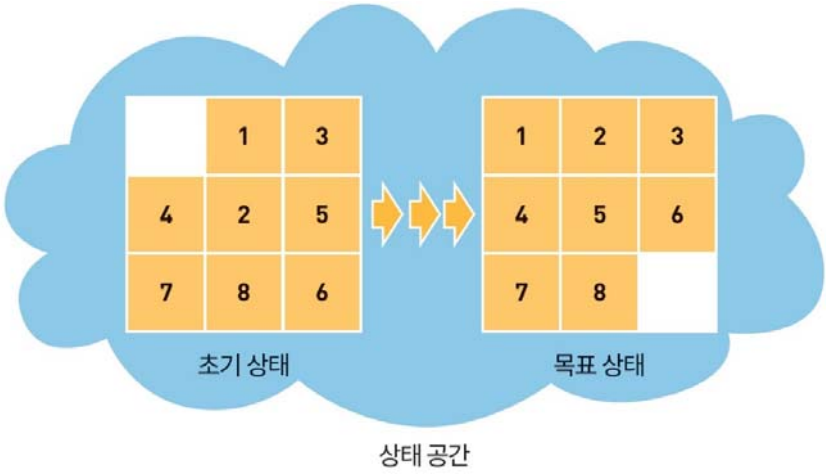


그림 2-2 상태, 상태 공간, 연산자

8-puzzle



8-puzzle



8-puzzle에서의 연산자

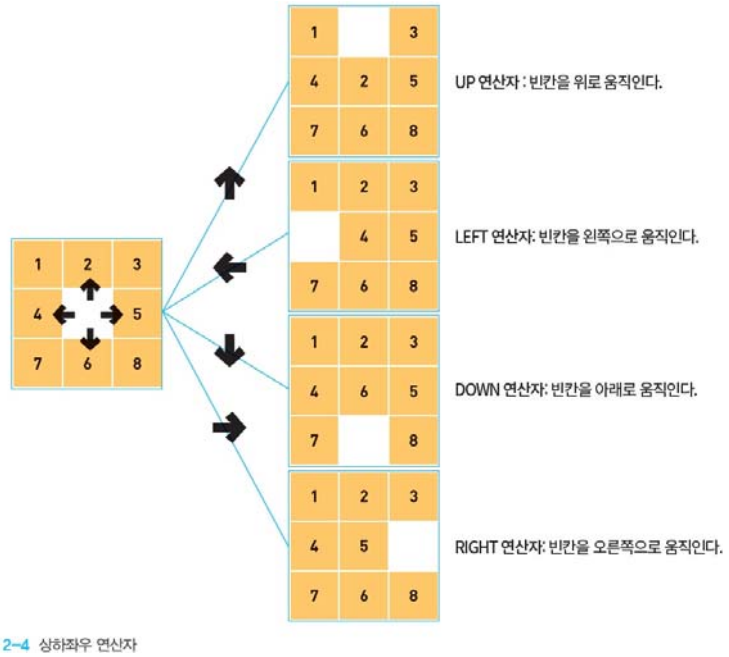


그림 2-4 상하좌우 연산자

8-puzzle 에서의 상태 공간

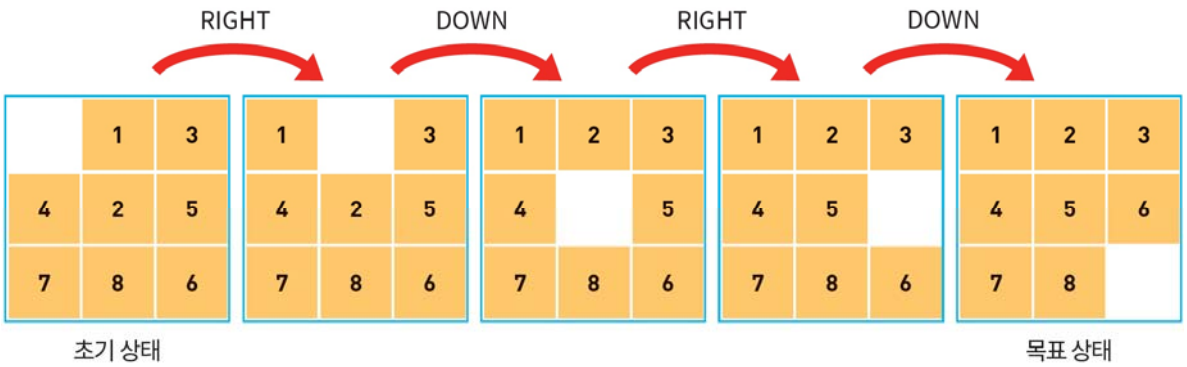


그림 2-5 연산자를 사용하여 이동하는 8-퍼즐 예제

Lab: 하노이 탑

- 상태?
- 연산자?

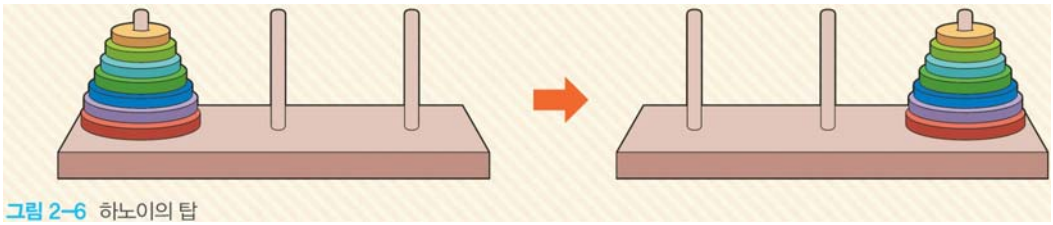


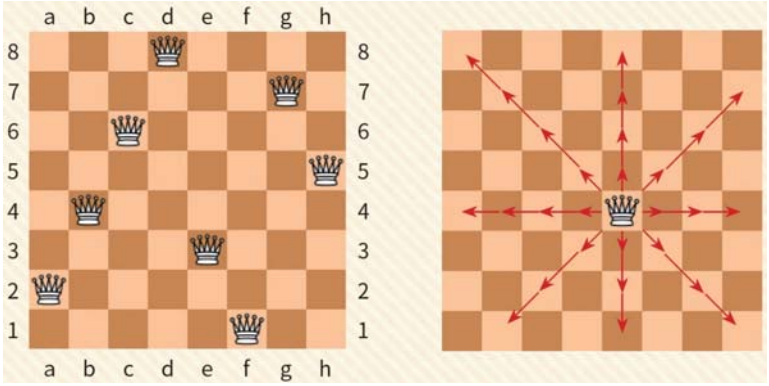
그림 2-6 하노이의 탑

Lab: 하노이 탑

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

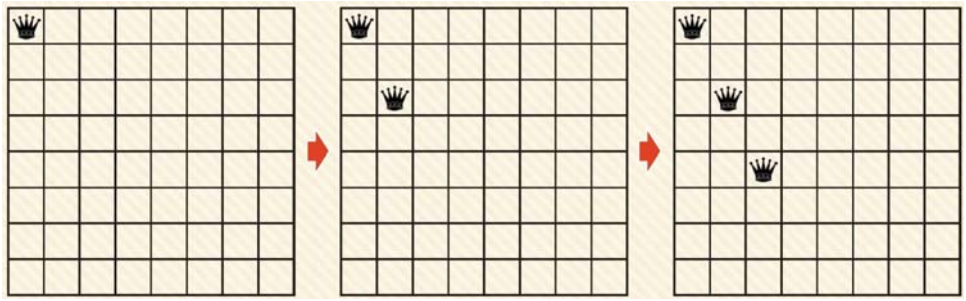
Lab:N-queen

- 상태?
- 연산자?



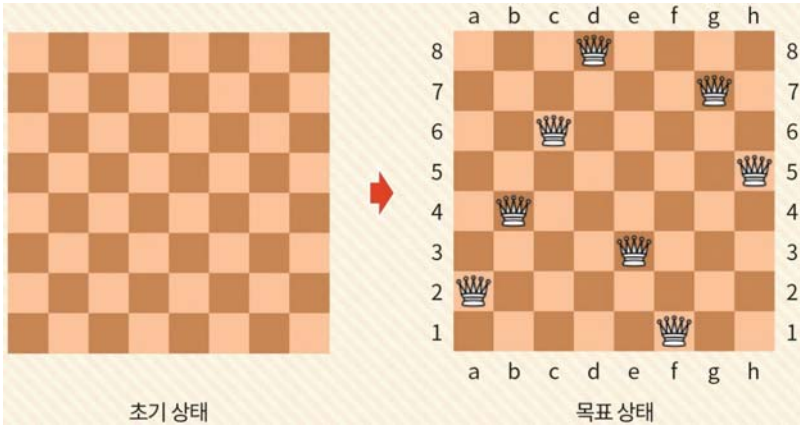
Lab:N-queen

- 각각의 퀸을 정해진 열에서만 움직이게 한다.



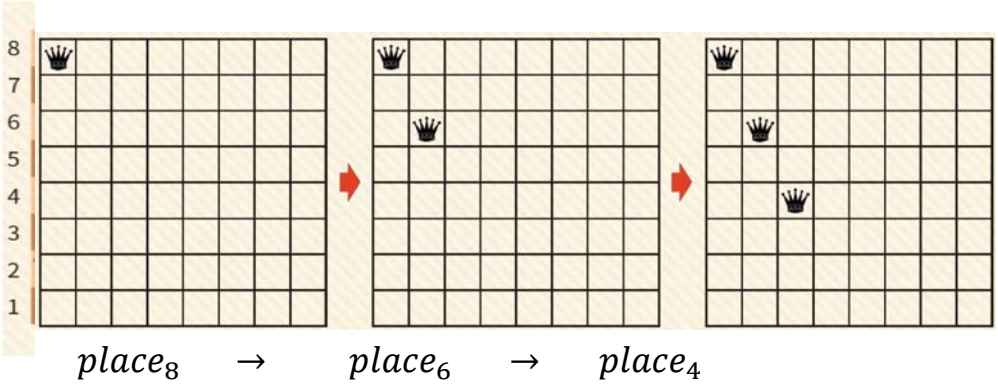
Lab:N-queen

- 초기 상태와 목표 상태



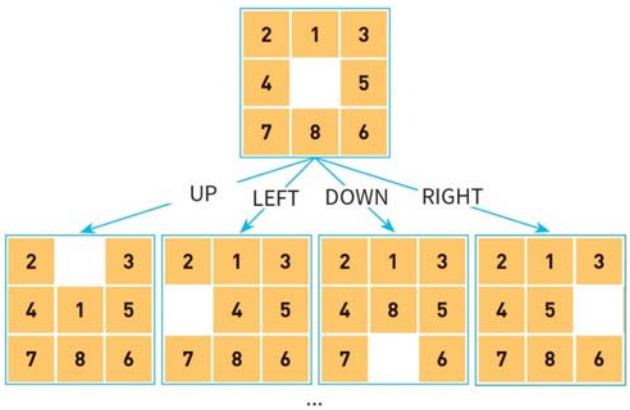
Lab:N-queen

- 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!-

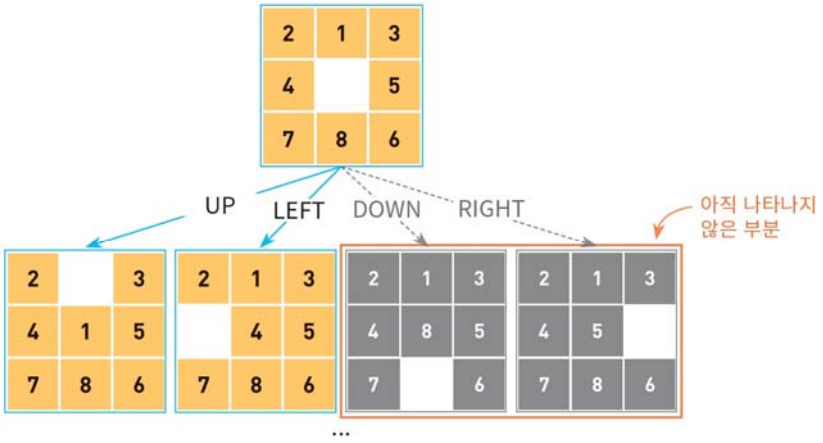
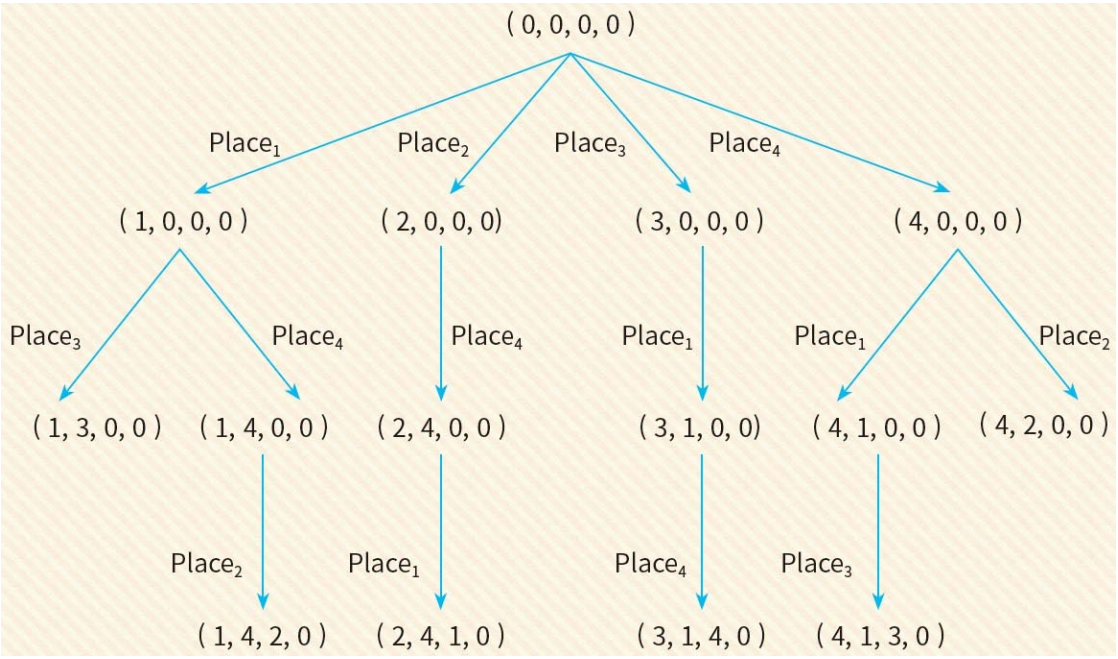


그림 2-9 탐색 트리의 노드는 동적으로 생성된다.

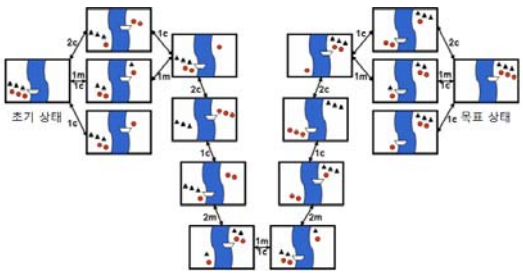
Lab: 4-queen 문제 탐색 트리의 일부



- 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)
  - **상태공간**에서 각 **행동**에 따른 **상태의 변화**를 나타낸 그래프
    - 노드 : 상태
    - 링크 : 행동
  - **선교사-식인종 문제**



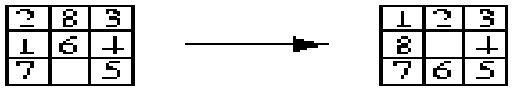
▲ m: 선교사
● c: 식인종

  - 일반적인 문제에서는 **상태공간**이 **매우 큼**
    - 미리 상태 공간 그래프를 만들기 어려움
    - 탐색과정에서 그래프 생성

- **해(solution)**
  초기상태에서 목표 상태로의 경로(path)

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

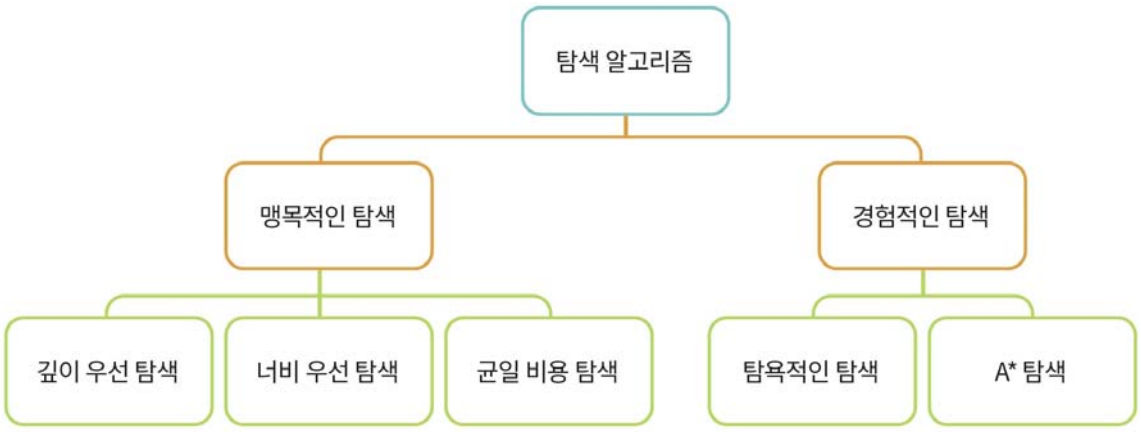


그림 2-10 탐색의 종류

깊이 우선 탐색(DFS)

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

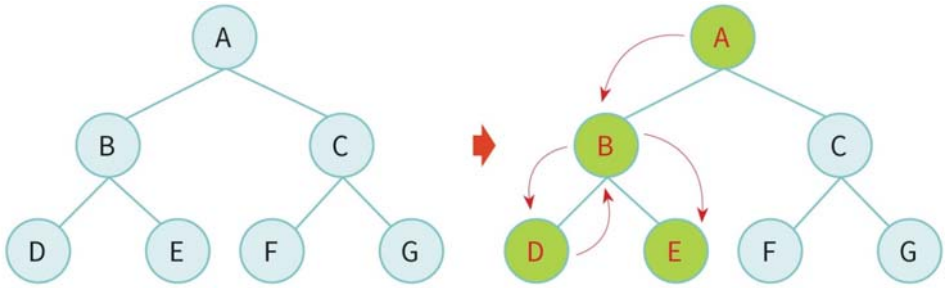


그림 2-11 깊이 우선 탐색

깊이 우선 탐색(8-puzzle)

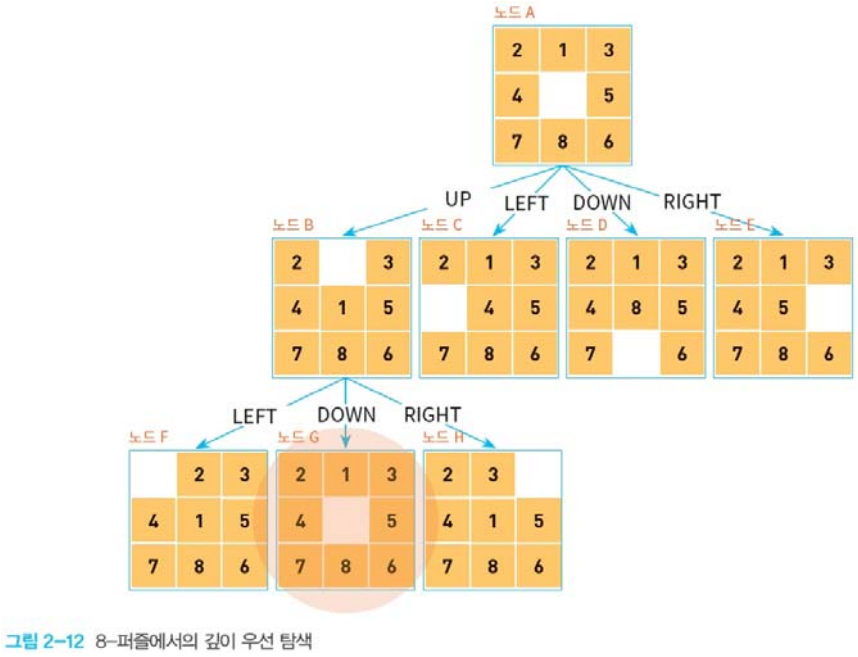
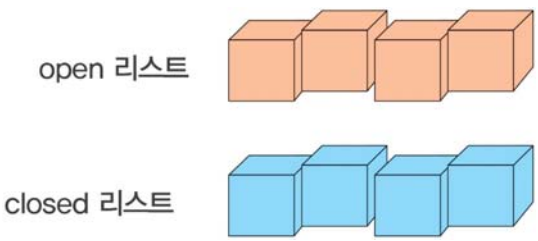


그림 2-12 8-퍼즐에서의 깊이 우선 탐색

OPEN CLOSED 리스트

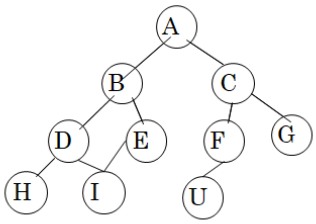
- 탐색에서는 중복된 상태를 막기 위하여 다음과 같은 2개의 리스트를 사용한다.
  - **OPEN** 리스트: 확장은 되었으나 아직 탐색하지 않은 상태들이 들어 있는 리스트
  - **CLOSED** 리스트: 탐색이 끝난 상태들이 들어 있는 리스트



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: 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!
```

너비 우선 탐색(BFS)

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

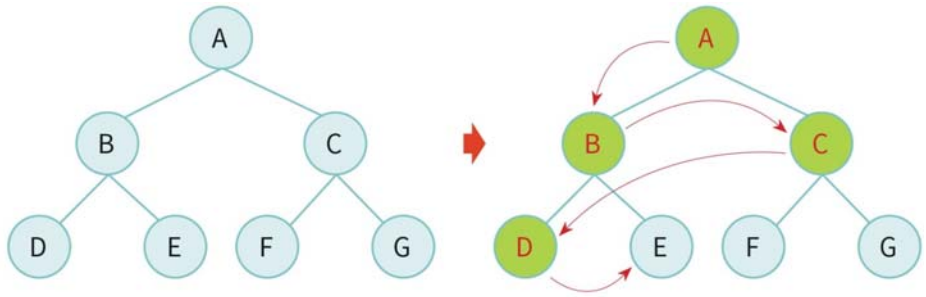
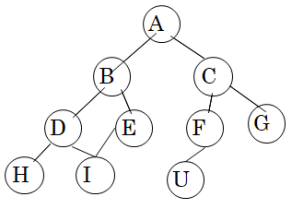


그림 2-13 너비 우선 탐색

BFS 알고리즘

```
breadth_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

- 실습: 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

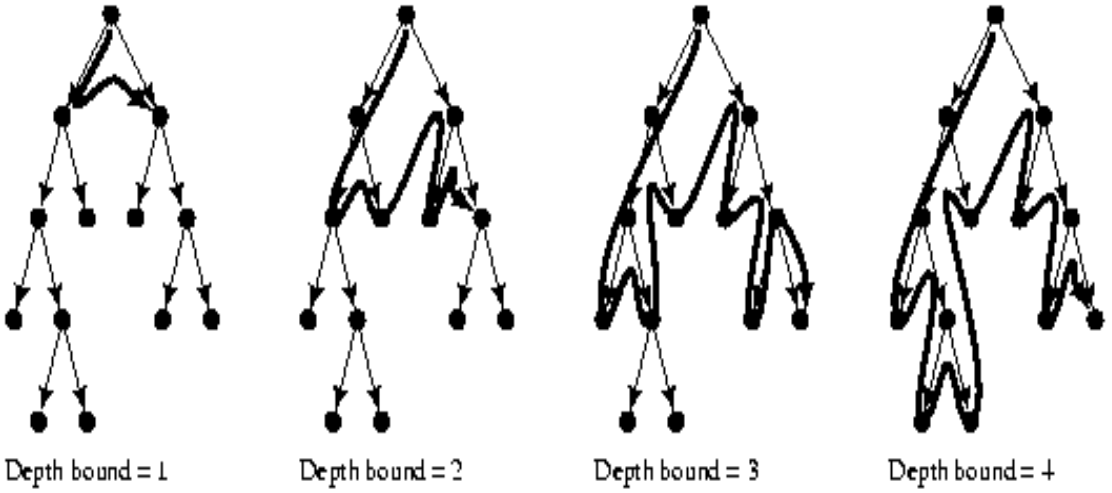
```
# open 리스트
open_queue = [ ]
open_queue.append(State(puzzle, goal))

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
```

state의 board와 각 queue의
State.board를 비교해야함

실행 결과

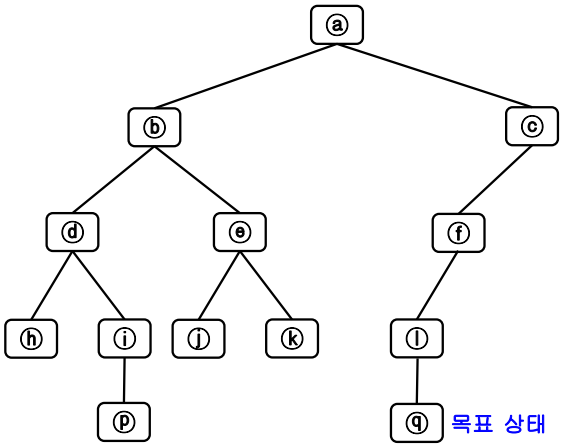
```
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
------------------
...
...
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
------------------
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
------------------
탐색 성공
```


반복적 깊이 증가 (iterative deepening)



맹목적 탐색

- 반복적 깊이증가 탐색(iterative-deepening search)
  - **깊이 한계**가 있는 깊이 **우선 탐색**을 **반복적**으로 적용



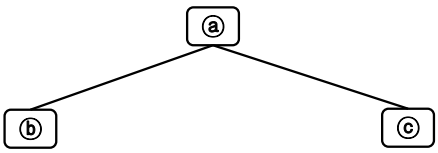
맹목적 탐색

- 반복적 깊이증가 탐색(iterative-deepening search)
  - **깊이 한계**가 있는 **깊이 우선 탐색**을 **반복적**으로 적용

깊이 0: ⓐ

맹목적 탐색

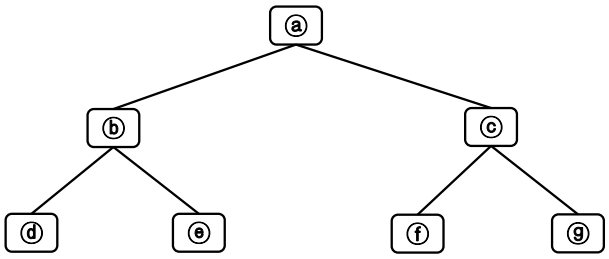
- 반복적 깊이증가 탐색(iterative-deepening search)
  - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ

맹목적 탐색

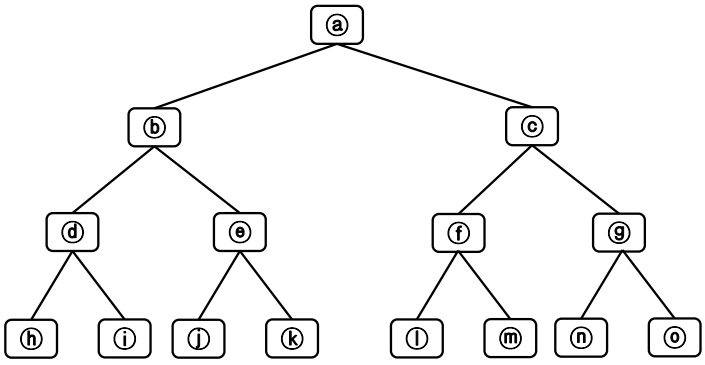
- 반복적 깊이증가 탐색(iterative-deepening search)
  - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ
깊이 2: ⓐ,ⓑ,ⓓ,ⓔ,ⓒ,ⓕ,ⓖ

맹목적 탐색

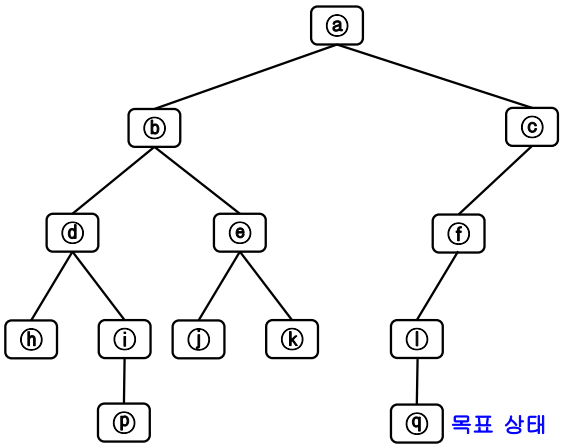
- 반복적 깊이증가 탐색(iterative-deepening search)
  - 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용



깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ
깊이 2: ⓐ,ⓑ,ⓓ,ⓔ,ⓒ,ⓕ,ⓖ
깊이 3:
ⓐ,ⓑ,ⓓ,ⓗ,ⓘ,ⓔ,ⓙ,ⓚ,ⓒ,ⓕ,ⓛ,ⓜ,ⓖ,ⓝ,ⓞ

맹목적 탐색

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



깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ
깊이 2: ⓐ,ⓑ,ⓓ,ⓔ,ⓒ,ⓕ,ⓖ
깊이 3:
ⓐ,ⓑ,ⓓ,ⓗ,ⓘ,ⓔ,ⓙ,ⓚ,ⓒ,ⓕ,ⓛ,ⓜ,ⓖ,ⓝ,ⓞ
깊이 4: ⓐ,ⓑ,ⓓ,ⓗ,ⓘ,ⓟ,ⓔ,ⓙ,ⓚ,ⓒ,ⓕ,ⓛ,ⓠ

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 때가지 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 d에 있는 경우
  - 너비우선 탐색으로 확장되는 노드 수

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

  - 깊이 j까지 깊이 우선 탐색으로 확장되는 노드 수

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

- 반복적 깊이증가 탐색

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

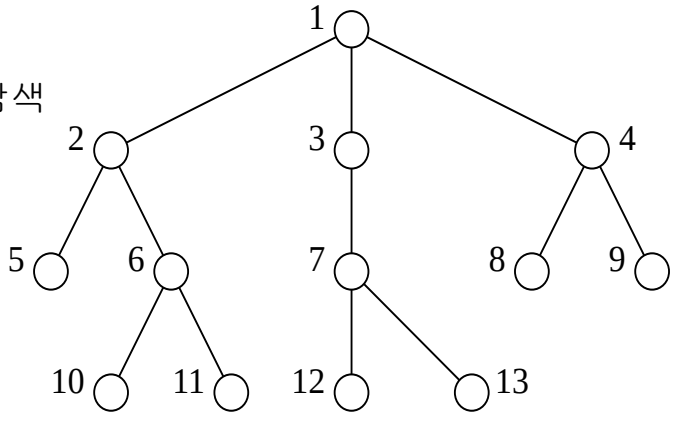
$$
\frac{N_{id}}{N_{bf}} = \frac{\dfrac{b^{d+2}-2b-bd+d+1}{(b-1)^2}}{\dfrac{b^{d+1}-1}{b-1}} = \frac{b^{d+2}-2b-bd+d+1}{\left(b^{d+1}-1\right)(b-1)} = \frac{b}{b-1}
$$

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 자식노드**를 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



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.

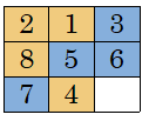
8-puzzle에서의 휴리스틱

예를 들어서 현재 상태와 목표 상태가 다음과 같다고 하자.

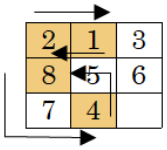
2	1	3
8	5	6
7	4	

1	2	3
4	5	6
7	8	

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



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



09 언덕 등반 기법

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

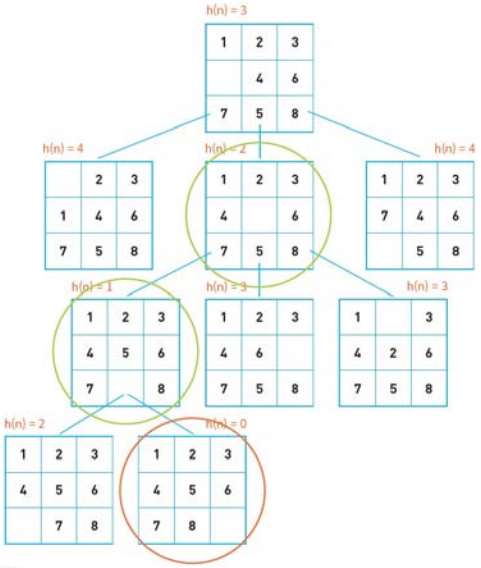
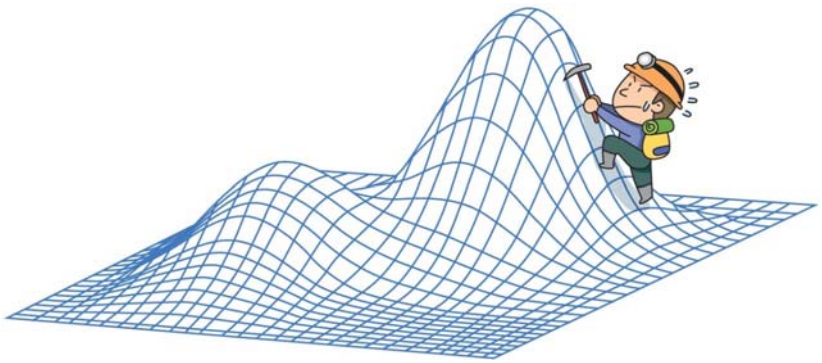


그림 2-14 언덕 등반 기법

언덕 등반 기법

- 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.

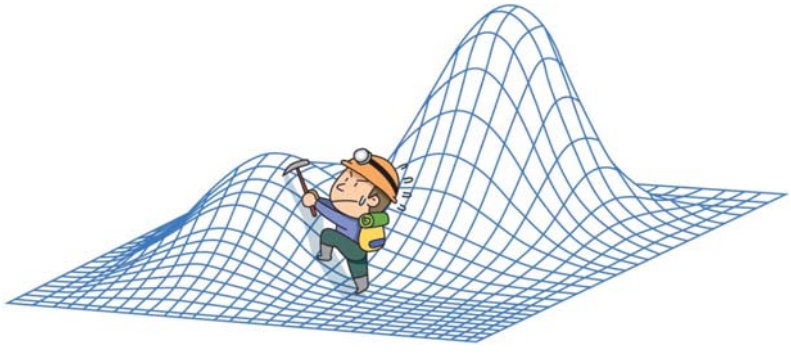


언덕 등반 기법 알고리즘

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).

지역 최소 문제

- 순수한 언덕 등반 기법은 오직 h(n) 값만을 사용한다(OPEN 리스트나 CLOSED 리스트도 사용하지 않는다 -> 과거 기억이 없음).
- 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

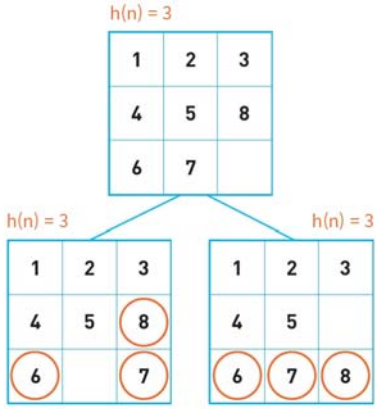


그림 2–15 지역 최소 문제

휴리스틱 탐색

- 평가 함수를 사용
  - 문제의 특성에 대한 정보인 휴리스틱에 따라 목표까지의 가장 좋은 경로상에 있다고 판단되는 노드를 우선 방문
- ***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. $\hat{f}(n)$ 값이 가장 작은 노드를 다음에 확장할 노드로 선택. 같은 값을 갖는 노드들은 무작위로 선택
3. 다음에 확장할 노드가 목표 노드이면 탐색을 종료

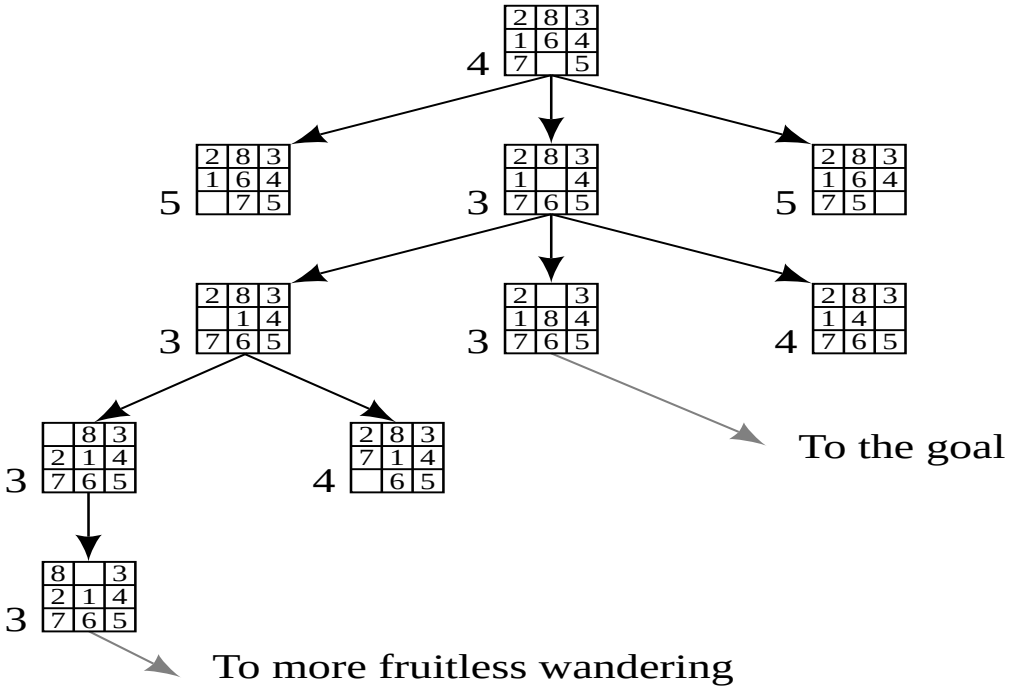
(ex) 8 퍼즐

$\hat{f}(n)$ = 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 n의 깊이에 대한 추정값

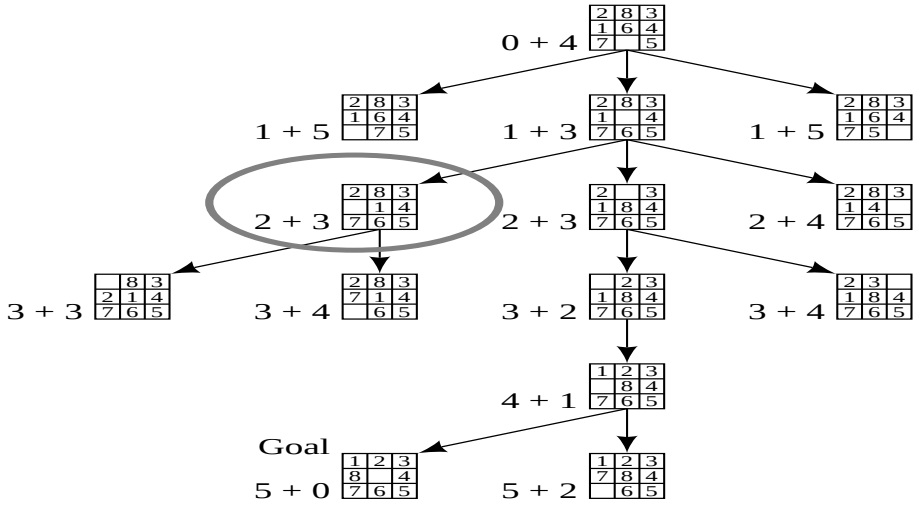
$\hat{h}(n)$: 노드 n에 대한 휴리스틱 평가값

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 노드에서목표 노드까지의 거리

8-puzzle 에서의 A* 알고리즘

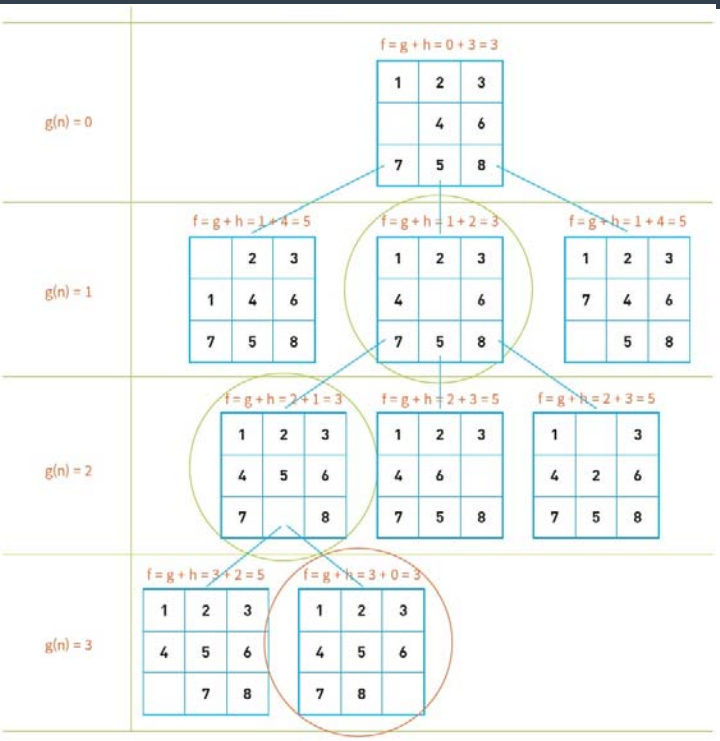


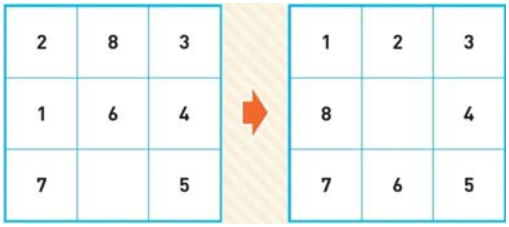
그림 2-17 8-퍼즐에서의 A* 알고리즘

A* 알고리즘

```
AStar_search()

open ← [시작노드]
closed ← [ ]
while open ≠ [ ] do
        X ← open 리스트에서 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
```

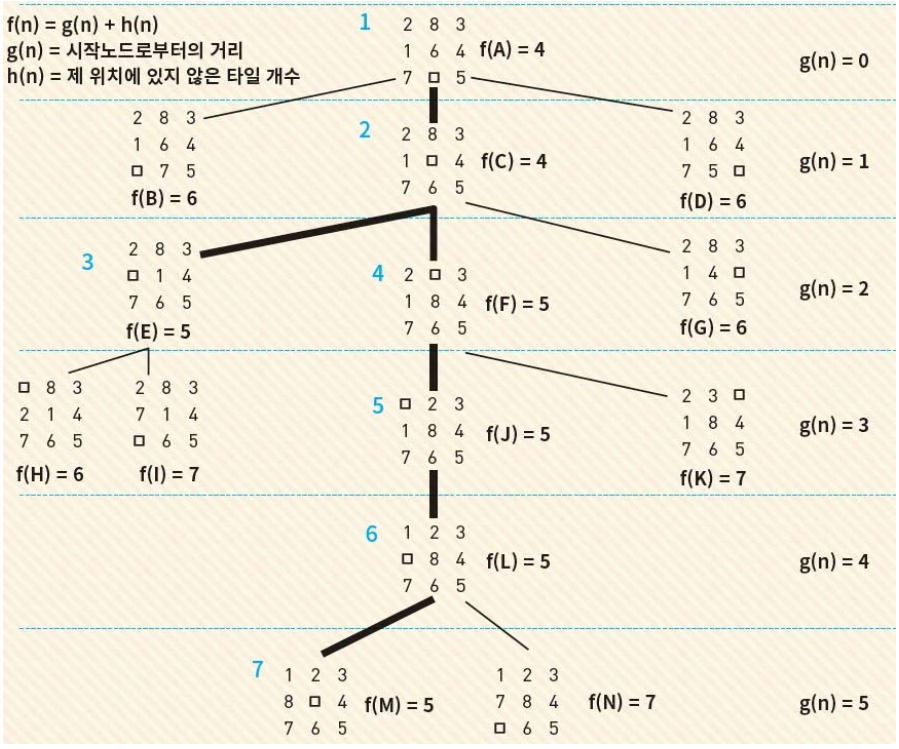
Lab: A* 알고리즘을 시뮬레이션

- 시작 상태와 목표 상태



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

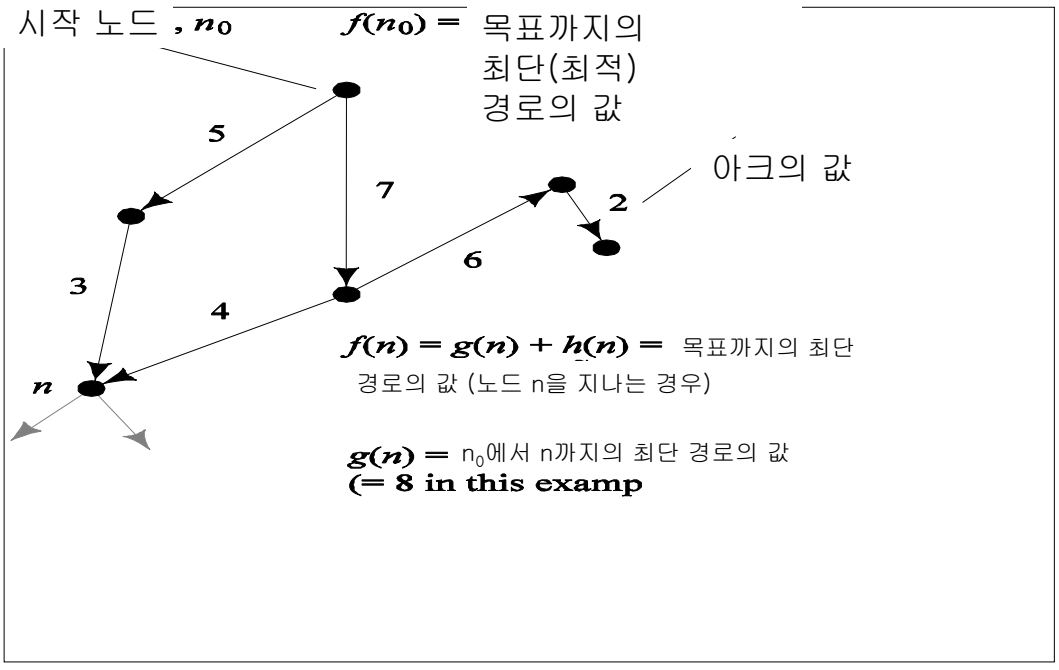
Lab: A* 알고리즘을 시뮬레이션



A* 알고리즘

- g(n) : 시작 노드 n_0에서 노드 n까지의 최단 경로의 값, $\hat{g}(n)$는 A*에 의하여 지금까지 발견된 노드 n까지의 경로 중에서 최단 경로의 값
- h(n) : 노드 n과 목표 노드 사이의 최단 경로의 실제 값, $\hat{h}(n)$은 h(n)의 추정값
- f(n) = g(n) + h(n)
  - n_0에서 목표 노드까지 노드 n을 통하여 갈 수 있는 모든 가능한 경로 중에서 최단 경로의 값
  - $f(n_0) = h(n_0)$ 는 n_0에서 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 n에 대하여,

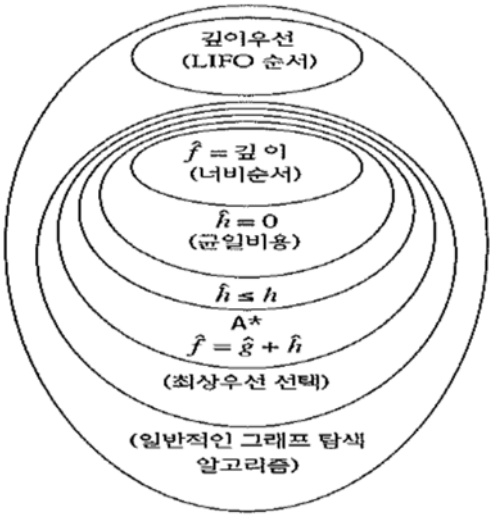
$$\hat{h}(n) \leq h(n)$$

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

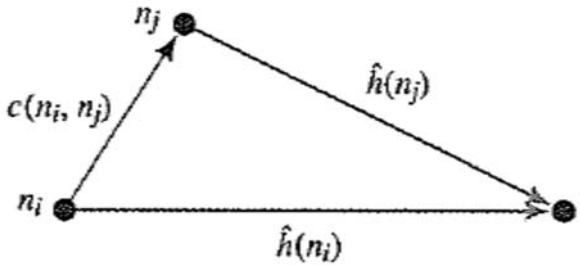
두 개의 서로 다른 A* 알고리즘, A^*_1와 A^*_2가 , 목표 외의 모든 노드에 대하여 $\hat{h}_1 < \hat{h}_2$ 인 점만 다르다면, A^*_2 가 A^*_1보다 정보가 많다 (*more informed*)고 한다.

A^*_2가 A^*_1보다 정보가 많으면(*more informed*), n_0에서 목표 노드까지 경로가 있는 모든 그래프에 대해서, 탐색이 종료되었을 때 A^*_2에 의해 확장된 노드는 A^*_1에 의해서도 확장된다.

탐색 알고리즘 사이의 관계



A*의 일관성(단조성) 조건



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

모든 노드 쌍이 위의 조건을 만족하면 일관성 조건을 만족한다고 함

A*의 일관성 조건 = A*의 단조성 조건

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

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

A*의 일관성(단조성) 조건

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

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

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

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

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

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

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

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

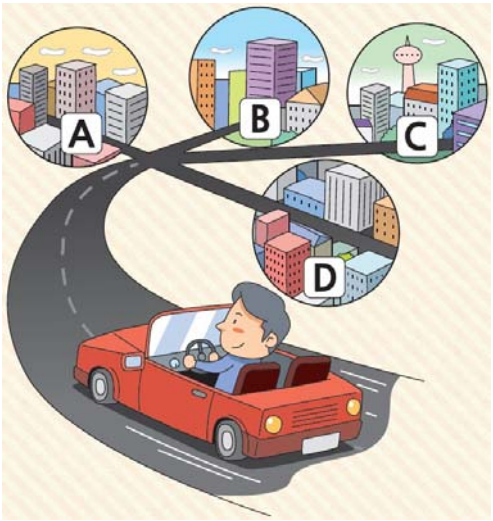
```
closed_queue = [ ]
moves = 0
while not open_queue.empty():

  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 경로는 무엇인가? " 이다.



과제

- N-queen 문제를 A* 탐색 방법을 사용하여 파이썬으로 구현
  - 정수 N을 입력받아 N개의 queen이 서로 충돌하지 않도록 N*N 보드에 배치하는 문제를 A* 탐색 방법을 사용하여 구현
  - 제출물: 학번과 이름이 소스코드와 출력 화면에 모두 표기된 소스코드 파일과 출력 캡처 화면 파일

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