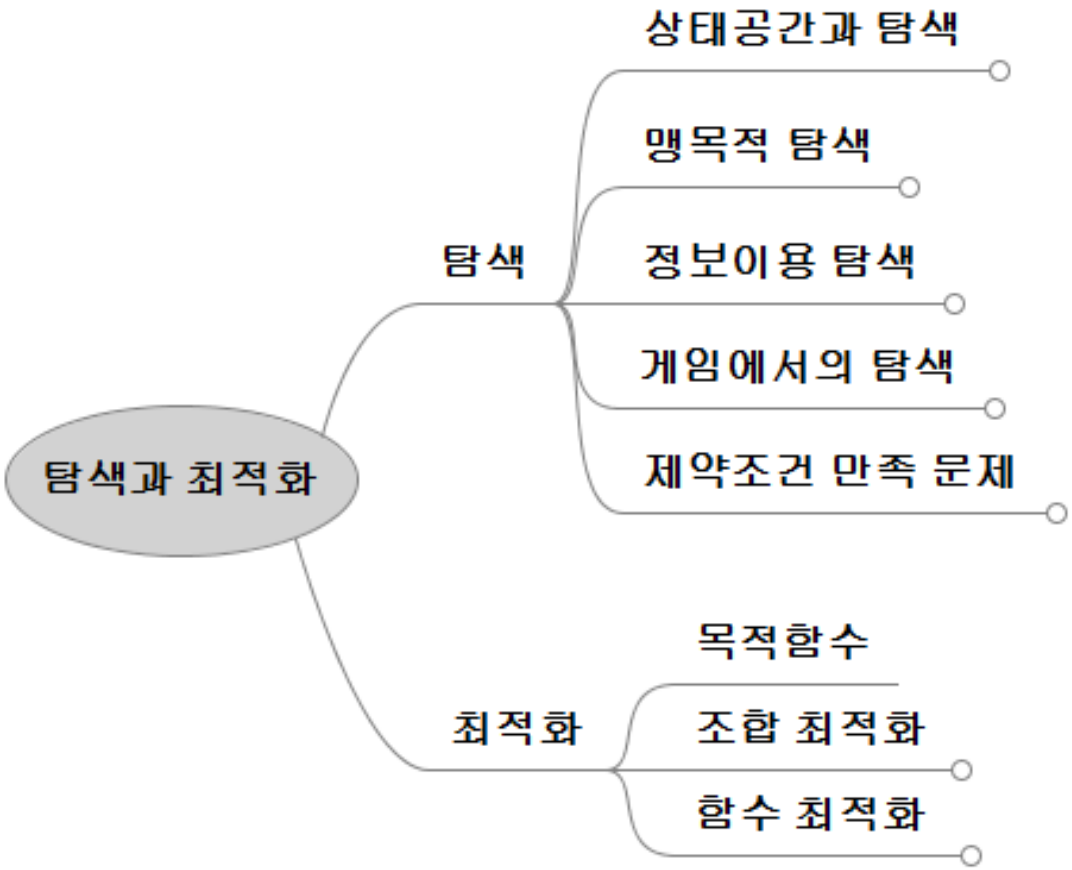
탐색과 최적화 - 1

이건명

충북대학교 소프트웨어학과

인공지능 : 튜링 테스트에서 딥러닝까지


탐색과 최적화
탐색
상태공간과 탐색
맹목적 탐색
정보이용 탐색
게임에서의 탐색
제약조건 만족 문제
최적화
목적함수
조합 최적화
함수 최적화

학습 내용

- 상태공간과 탐색에 대해서 알아본다.
- 맹목적 탐색 기법인 깊이 우선 탐색, 너비 우선 탐색, 반복적 깊이심화 탐색에 대해서 살펴본다.
- 휴리스틱 탐색 기법인 언덕 오르기 탐색, 최선 우선 탐색, 빔 탐색, A* 알고리즘에 대해서 알아본다.

1. 상태 공간과 탐색

❖ **탐색** (探索, search)

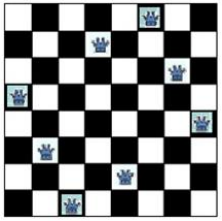
- 문제의 **해(solution)**이 될 수 있는 것들의 **집합**을 **공간**(space)으로 간주하고, 문제에 대한 **최적의 해**를 찾기 위해 공간을 **체계적으로 찾아 보는 것**

❖ **탐색문제의 예**

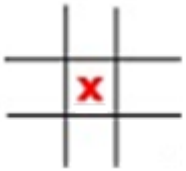
- **선교사-식인종 강건너기 문제**

- **8-퍼즐** 문제

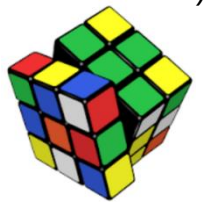
- **8-퀸**(queen) 문제



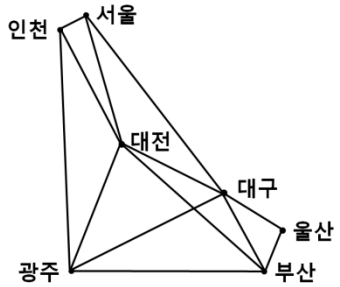
- **틱-택-토**(tic-tac-toe)



- **루빅스큐브** (Rubik's cube)



- **순회 판매자 문제 (**traveling salesperson problem, TSP**)**



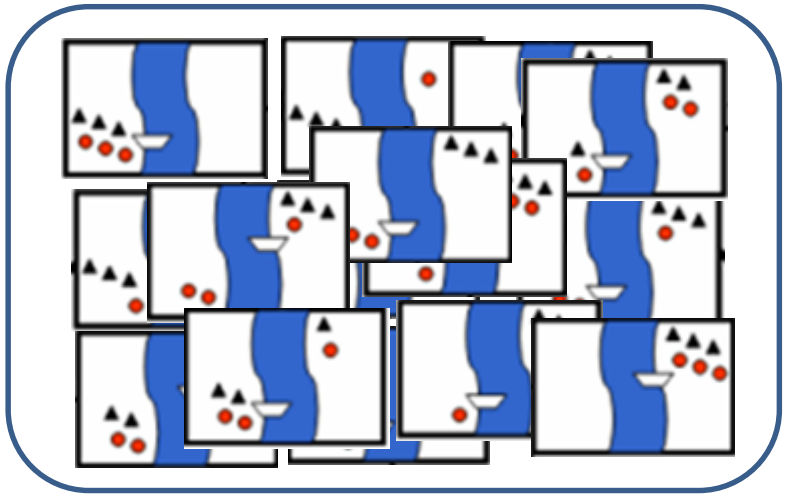
- **해(解, solution)**
  일련의 동작으로 구성되거나 **하나의 상태**로 구성

상태 공간과 탐색

❖ **상태(state)**
- 특정 시점에 **문제의 세계**가 처해 있는 모습


❖ **세계(world)**
- 문제에 포함된 **대상들**과 이들의 **상황**을 포괄적으로 지칭

❖ **상태 공간(state space)**
- 문제 해결 과정에서 **초기 상태로부터 도달할 수 있는 모든 상태들의 집합**
- 문제의 **해가 될 가능성**이 있는 **모든 상태들의 집합**



- **초기 상태(initial state)**

  - 문제가 주어진 시점의 시작 상태

- **목표 상태(goal state)**

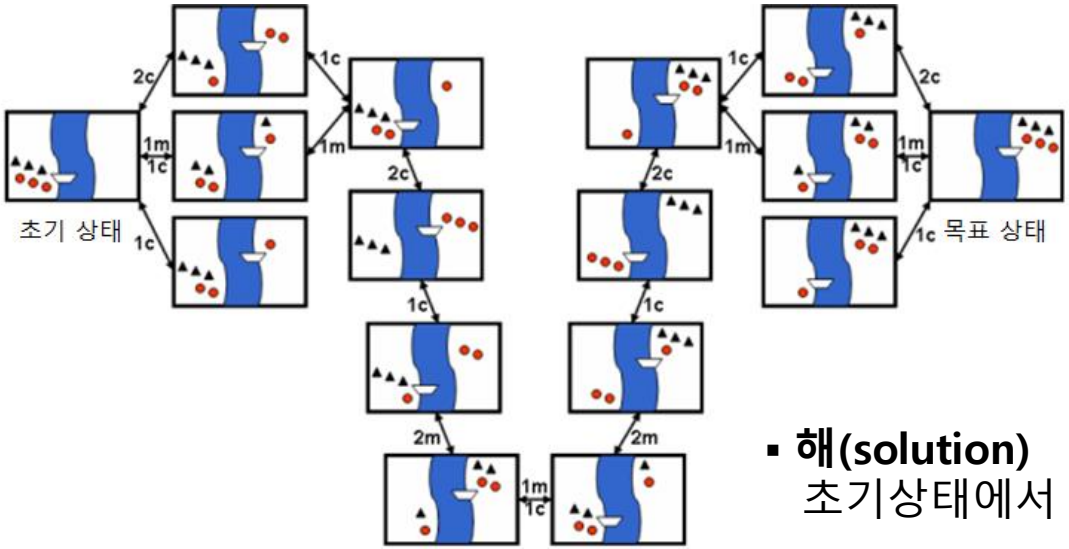
  - 문제에서 원하는 최종 상태

상태 공간과 탐색

❖ **상태 공간 그래프**(state space graph)

- **상태공간**에서 각 **행동**에 따른 **상태의 변화**를 나타낸 그래프
  - 노드 : 상태
  - 링크(에지) : 행동

- **선교사-식인종 문제**



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

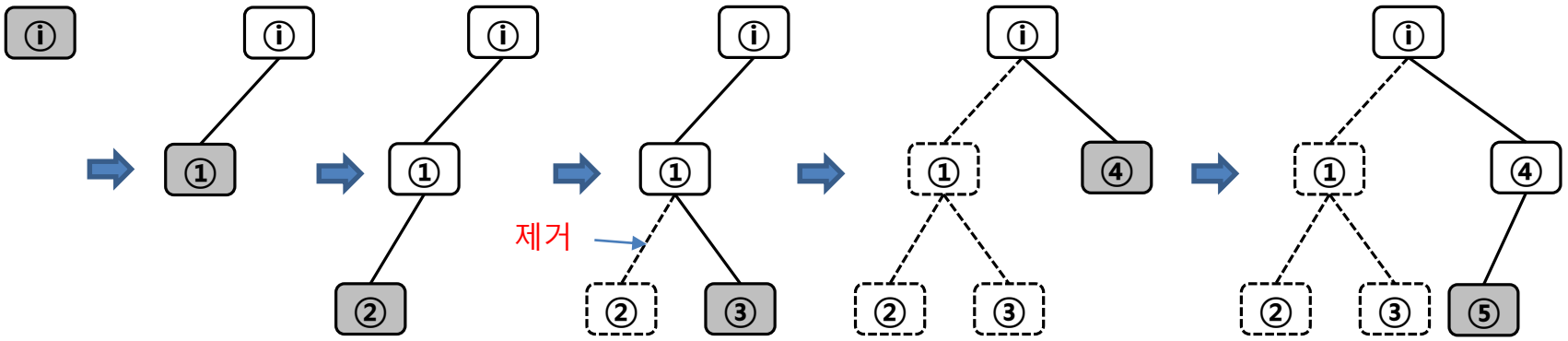
- 일반적인 문제에서는 **상태공간**이 **매우 큼**
  - 미리 상태 공간 그래프를 만들기 어려움
  - **탐색과정에서 그래프 생성**

2. 맹목적 탐색

❖ **맹목적 탐색(blind search)**

- **정해진 순서**에 따라 **상태 공간 그래프**를 **점진적으로 생성**해 가면서 해를 **탐색**하는 방법

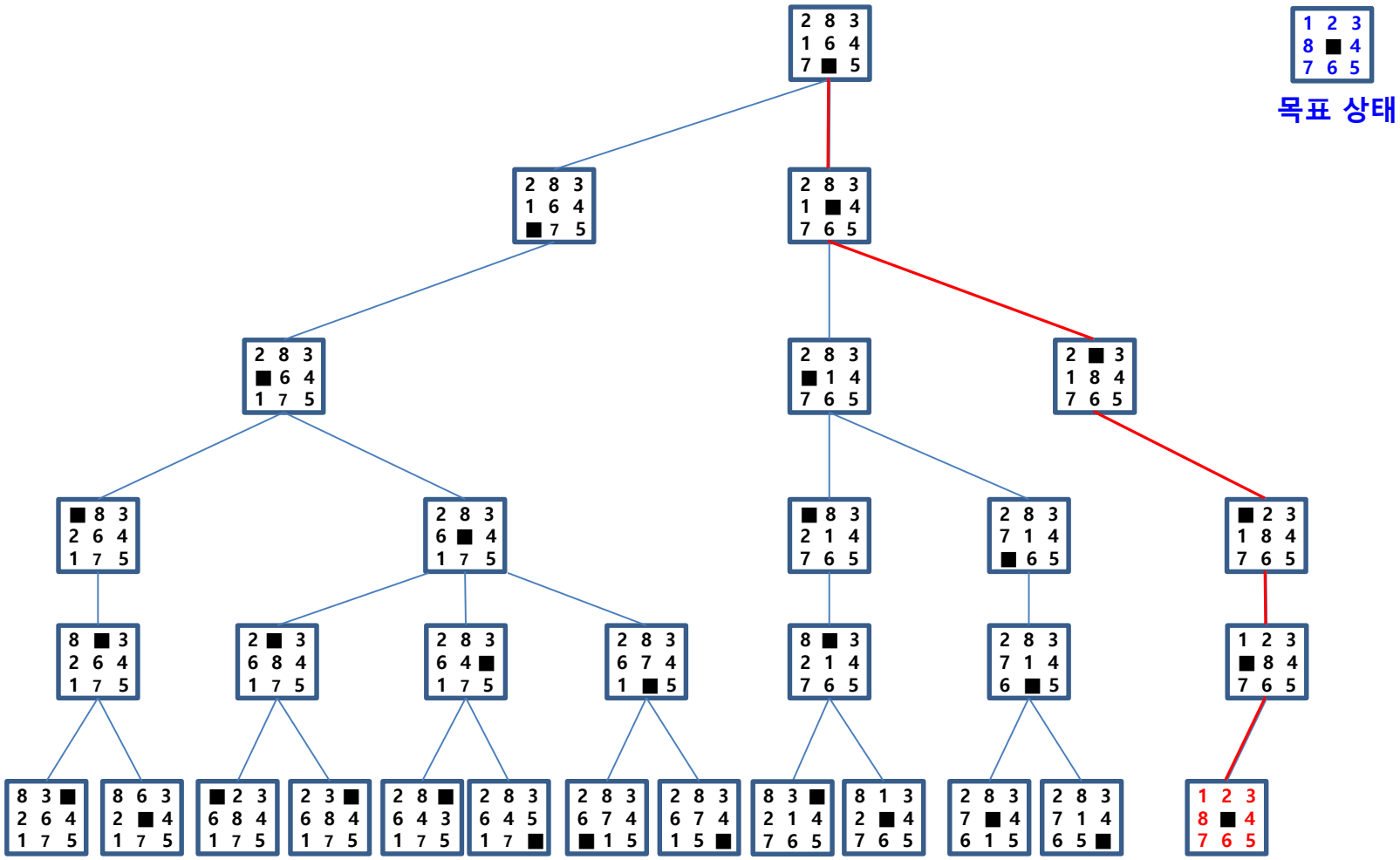
- **깊이 우선 탐색(depth-first search, DFS)**
  - 초기 노드에서 시작하여 **깊이 방향**으로 탐색
  - 목표 노드에 도달하면 종료
  - 더 이상 진행할 수 없으면, **백트랙킹**(backtracking, 되짚어가기)
  - 방문한 노드는 재방문하지 않음



맹목적 탐색

❖ 8-퍼즐 문제의 깊이 우선 탐색 트리

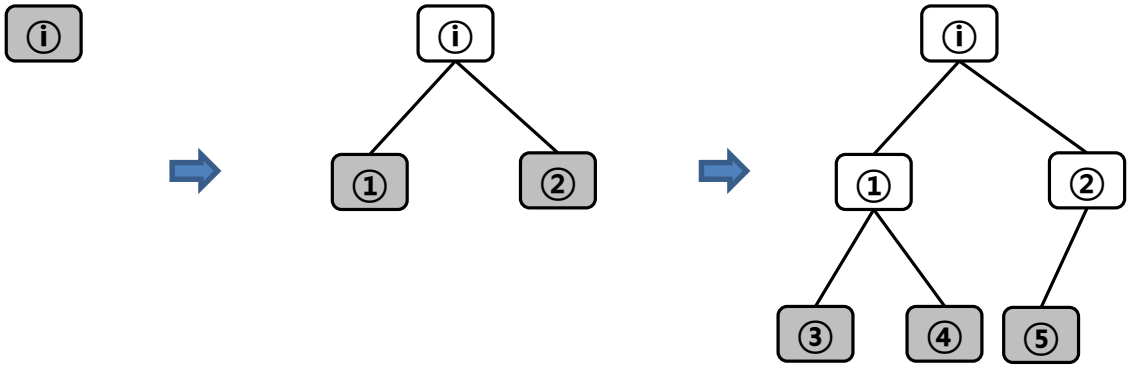
- 루트 노드에서 **현재 노드**까지의 **경로 하나만 유지**



맹목적 탐색

❖ **너비 우선 탐색**(breadth-first search, BFS)

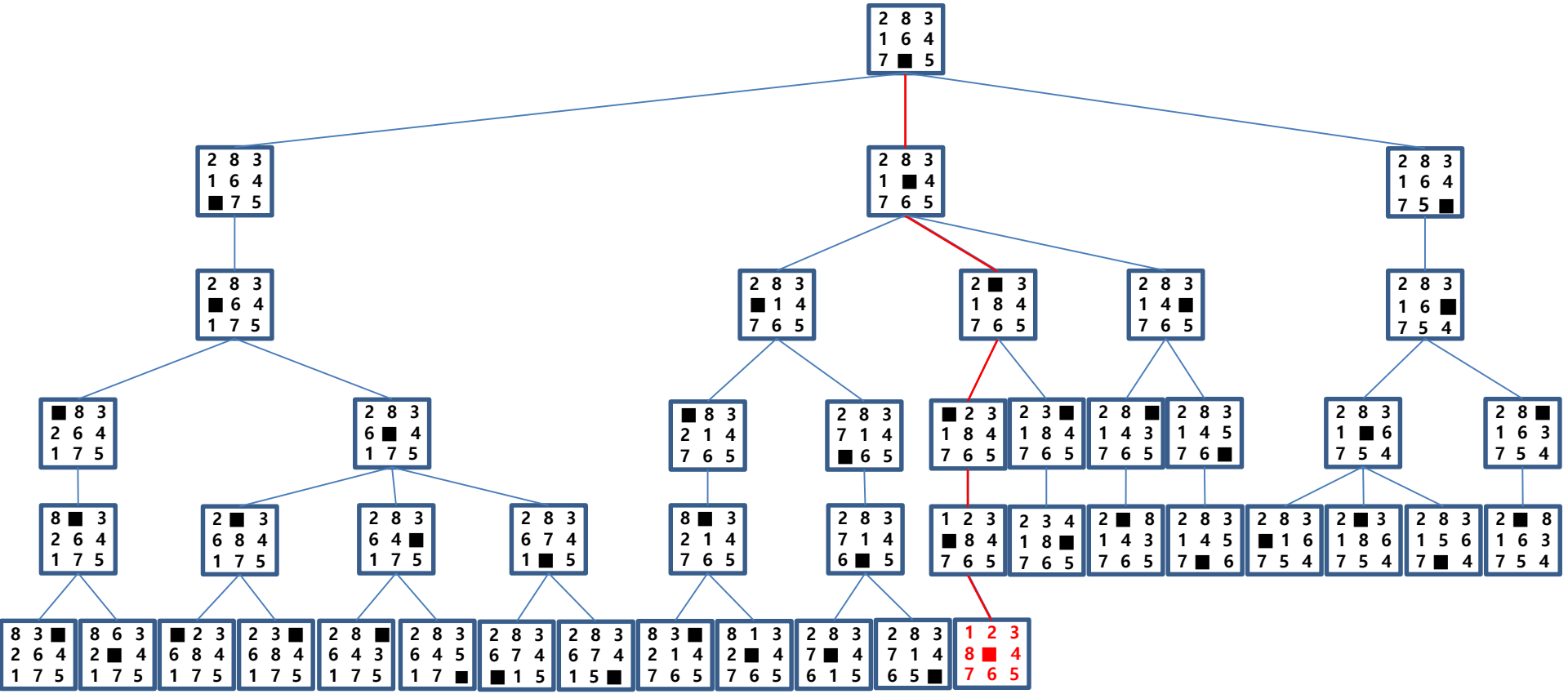
- 초기 노드에서 시작하여 **모든 자식 노드**를 **확장**하여 생성
- 목표 노드가 없으면 **단말노드**에서 다시 **자식 노드 확장**



맹목적 탐색

❖ **8-퍼즐 문제의 너비 우선 탐색 트리**

- **전체 트리**를 메모리에서 **관리**



맹목적 탐색

❖ **반복적 깊이심화 탐색**(iterative-deepening search)

- **깊이 한계**가 있는 **깊이 우선 탐색**을 **반복적**으로 적용

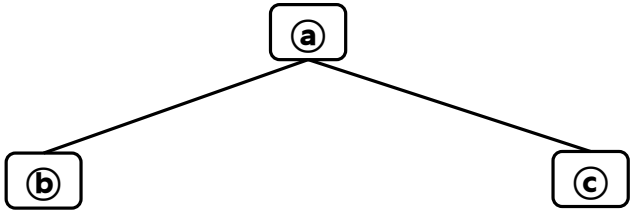
ⓐ

깊이 0: ⓐ

맹목적 탐색

❖ **반복적 깊이심화 탐색**(iterative-deepening search)

- **깊이 한계**가 있는 **깊이 우선 탐색**을 **반복적**으로 적용

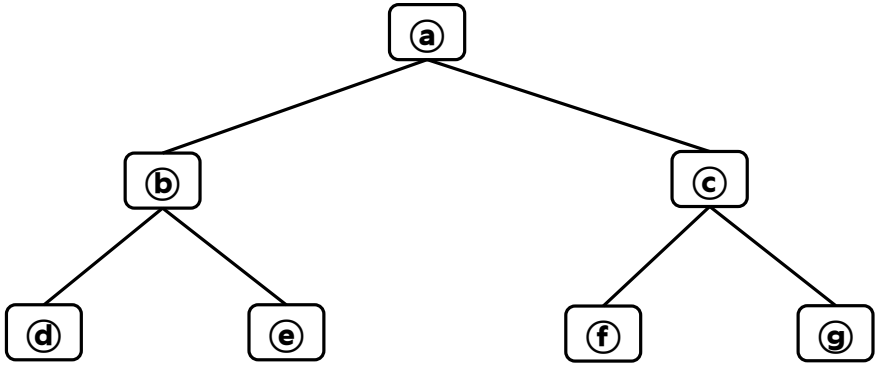


깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ

맹목적 탐색

❖ **반복적 깊이심화 탐색**(iterative-deepening search)

- **깊이 한계**가 있는 **깊이 우선 탐색**을 **반복적**으로 적용

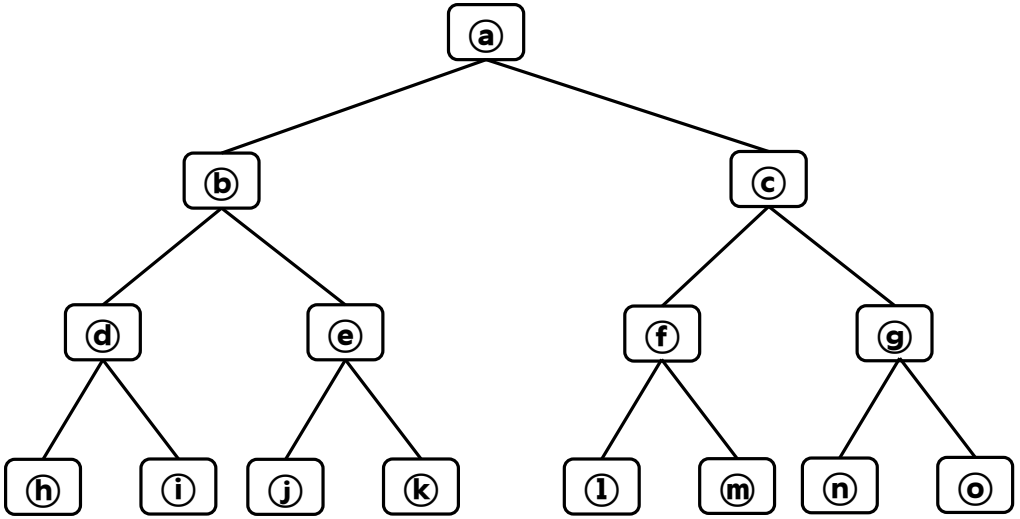


깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ
깊이 2: ⓐ,ⓑ,ⓓ,ⓔ,ⓒ,ⓕ,ⓖ

맹목적 탐색

❖ **반복적 깊이심화 탐색**(iterative-deepening search)

- **깊이 한계**가 있는 **깊이 우선 탐색**을 **반복적**으로 적용

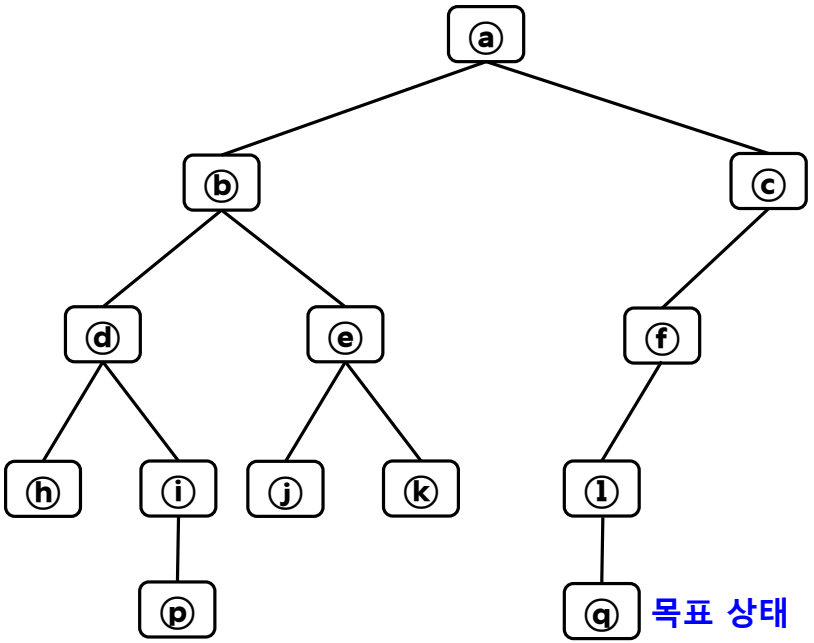


깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ
깊이 2: ⓐ,ⓑ,ⓓ,ⓔ,ⓒ,ⓕ,ⓖ
깊이 3: ⓐ,ⓑ,ⓓ,ⓗ,ⓘ,ⓔ,ⓙ,ⓚ,ⓒ,ⓕ,ⓛ,ⓜ,ⓖ,ⓝ,ⓞ

맹목적 탐색

❖ **반복적 깊이심화 탐색**(iterative-deepening search)

- **깊이 한계**가 있는 깊이 **우선 탐색**을 **반복적**으로 적용



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

맹목적 탐색

❖ **맹목적 탐색 방법의 비교**

- **깊이 우선 탐색**
  - **메모리 공간** 사용 **효율적**
  - 최단 경로 해의 탐색 보장 불가

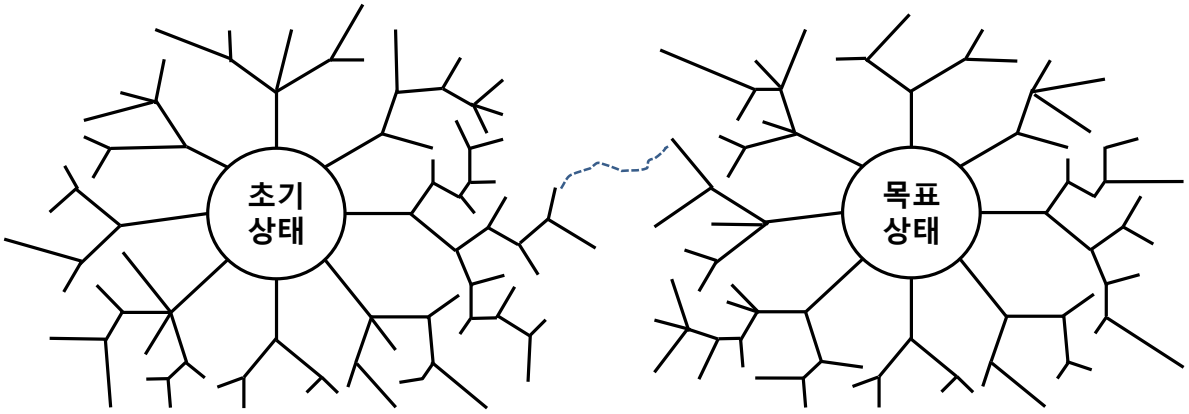
- **너비 우선 탐색**
  - **최단 경로 해**의 탐색 보장
  - 메모리 공간 사용 비효율

- **반복적 깊이심화 탐색**
  - **최단 경로 해**의 탐색 보장
  - **메모리 공간** 사용 **효율적**
  - 반복적인 깊이 우선 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

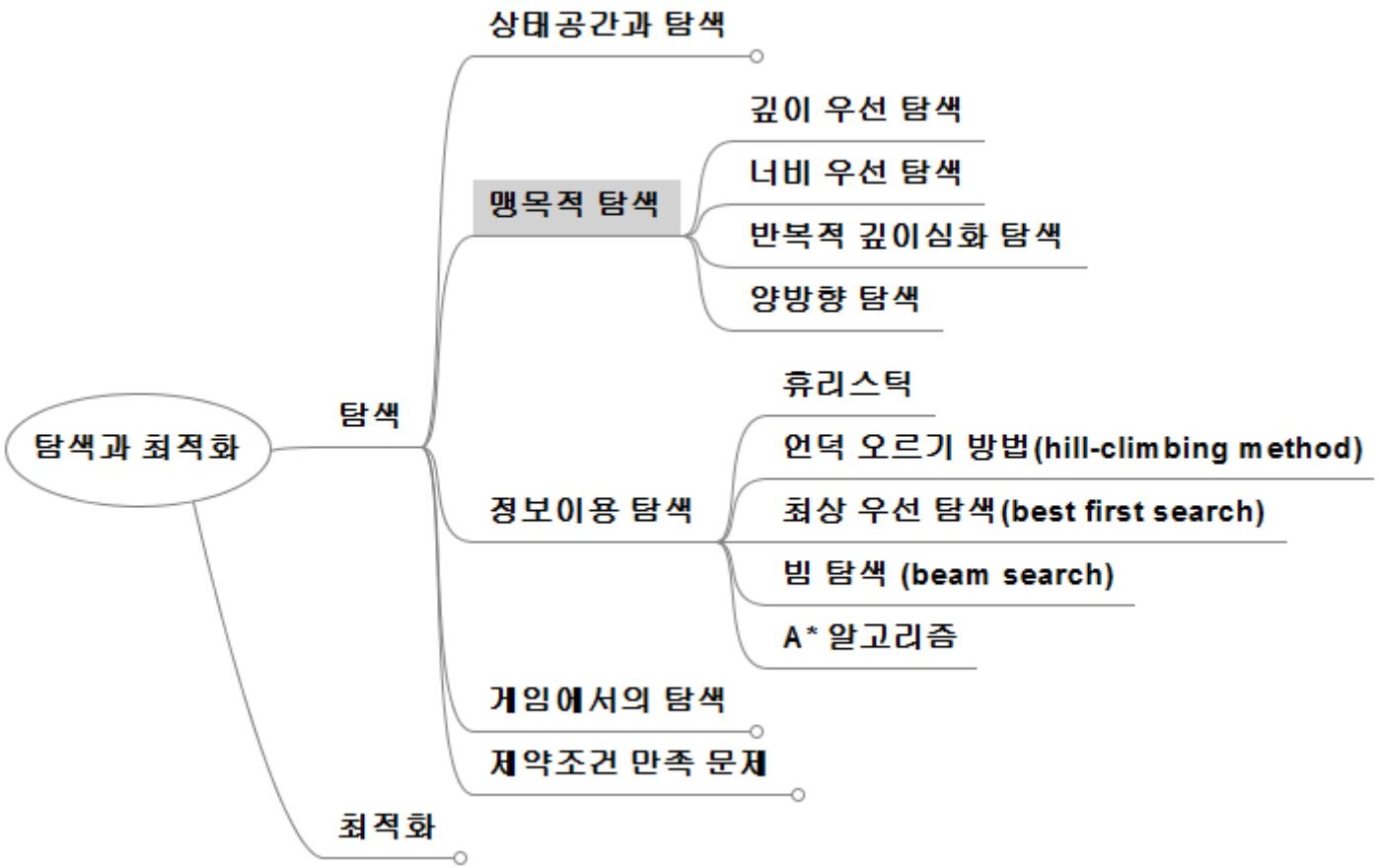
맹목적 탐색

❖ **양방향 탐색(bidirectional search)**

- **초기 노드**와 **목적 노드**에서 **동시**에 **너비 우선 탐색**을 진행
- 중간에 만나도록 하여 초기 노드에서 목표 노드로의 최단 경로를 찾는 방법



맹목적 탐색


탐색과 최적화
탐색
최적화
상태공간과 탐색
맹목적 탐색
깊이 우선 탐색
너비 우선 탐색
반복적 깊이심화 탐색
양방향 탐색
정보이용 탐색
휴리스틱
언덕 오르기 방법(hill-climbing method)
최상 우선 탐색(best first search)
빔 탐색 (beam search)
A* 알고리즘
게임에서의 탐색
제약조건 만족 문제

3. 정보이용 탐색

❖ **정보이용 탐색**(informed search)

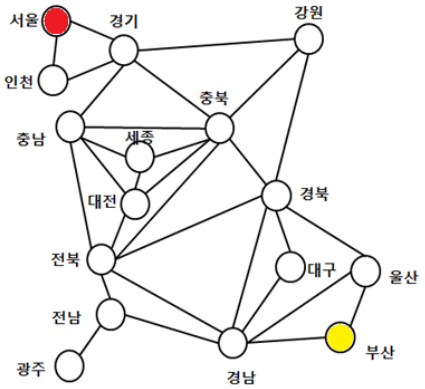
- **휴리스틱 탐색**(heuristic search)
- **언덕 오르기 방법**, **최상 우선 탐색**, **빔 탐색**, **A* 알고리즘** 등

- **휴리스틱**(heuristic)
  - 그리스어 Εὑρίσκω (Eurisko, 찾다, 발견하다)
  - 시간이나 정보가 불충분하여 합리적인 판단을 할 수 없거나, 굳이 체계적이고 합리적인 판단을 할 필요가 없는 상황에서 **신속하게 어림짐작하는 것**
  - 예.
    - **최단 경로 문제에서 목적지까지 남은 거리**
      - » 현재 위치에서 목적지(목표 상태)까지 **지도상의 직선 거리**

정보이용 탐색

❖ **휴리스틱 비용 추정의 예**

- **최단경로 문제**
  - 현재 위치에서 목적지까지 직선 거리



- **8-퍼즐 문제**
  - 제자리에 있지 않는 타일의 개수

2	8	3
1	6	4
7		5

현재 상태

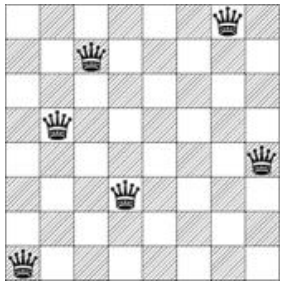
1	2	3
8		4
7	6	5

목표 상태

추정비용 : 4

- **8-퀸 문제**

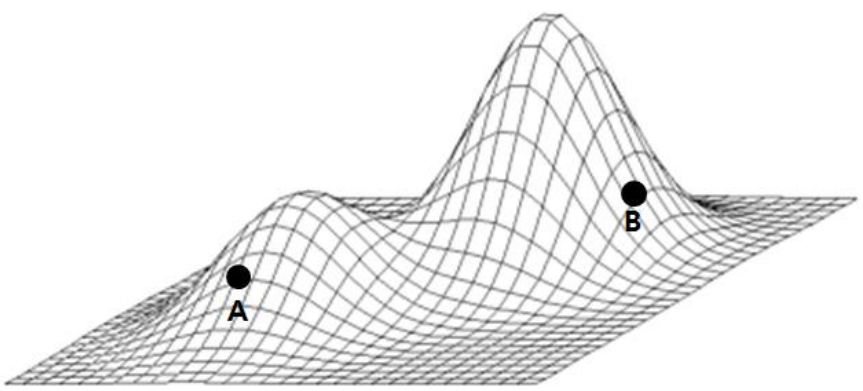
  충돌하는 회수



정보이용 탐색

❖ **언덕 오르기 방법**(hill climbing method)

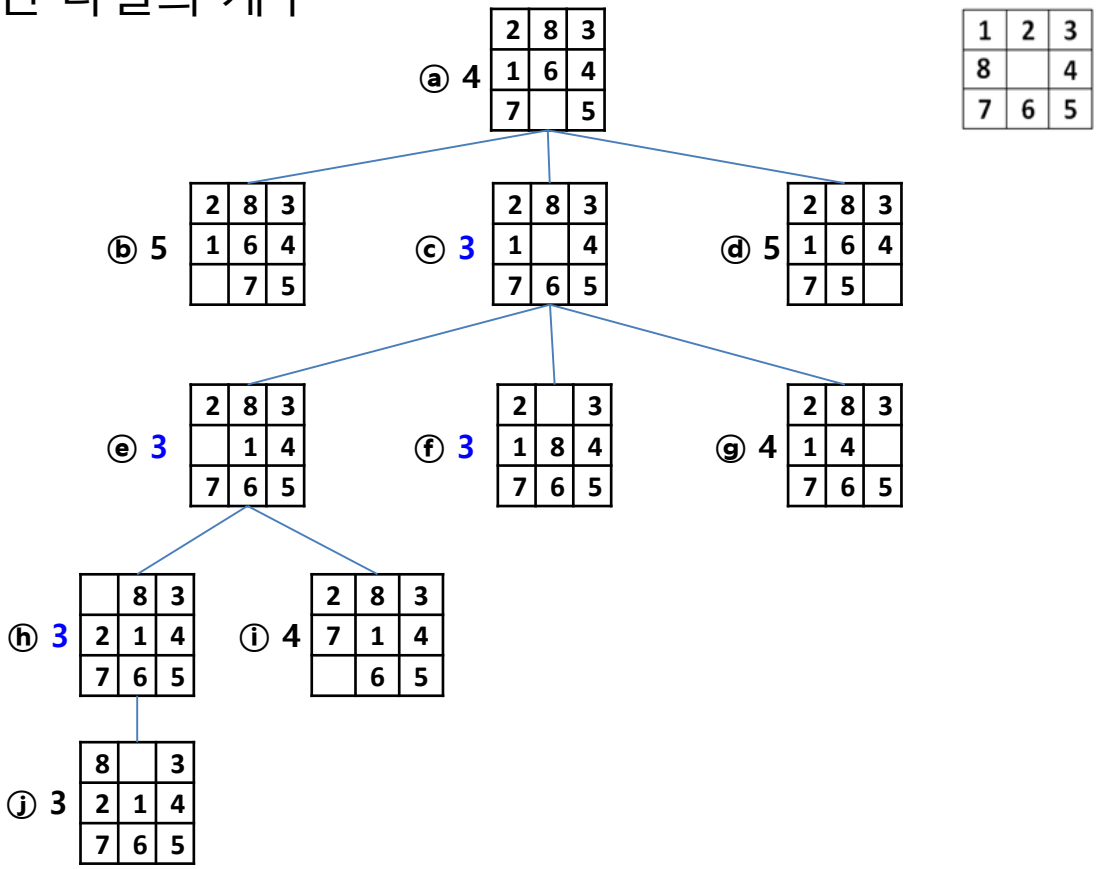
- **지역 탐색**(local search), **휴리스틱 탐색**(heuristic search)
- 현재 노드에서 휴리스틱에 의한 평가값이 **가장 좋은 이웃 노드** 하나를 **확장**해 가는 탐색 방법
- **국소 최적해**(local optimal solution)에 빠질 가능성



정보이용 탐색

❖ **최상 우선 탐색**(best-first search)

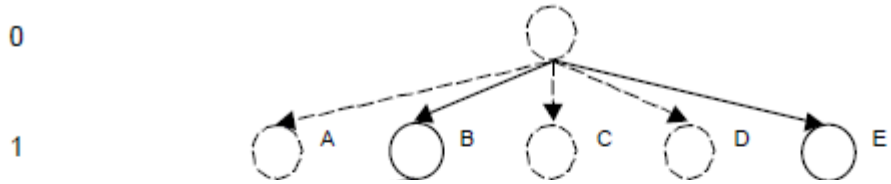
- 확장 중인 노드들 중에서 목표 노드까지 **남은 거리가 가장 짧은 노드**를 확장하여 탐색
- 남은 거리를 정확히 알 수 없으므로 **휴리스틱** 사용
  - 제자리가 아닌 타일의 개수



정보이용 탐색

❖ 빔 탐색(beam search)

- 휴리스틱에 의한 평가값이 우수한 **일정 개수의 확장 가능한 노드**만을 메모리에 **관리**하면서 **최상 우선 탐색**을 적용



정보이용 탐색

❖ A* 알고리즘

- **추정한 전체 비용** $\hat{f}(n)$을 **최소**로 하는 **노드**를 확장해 가는 방법

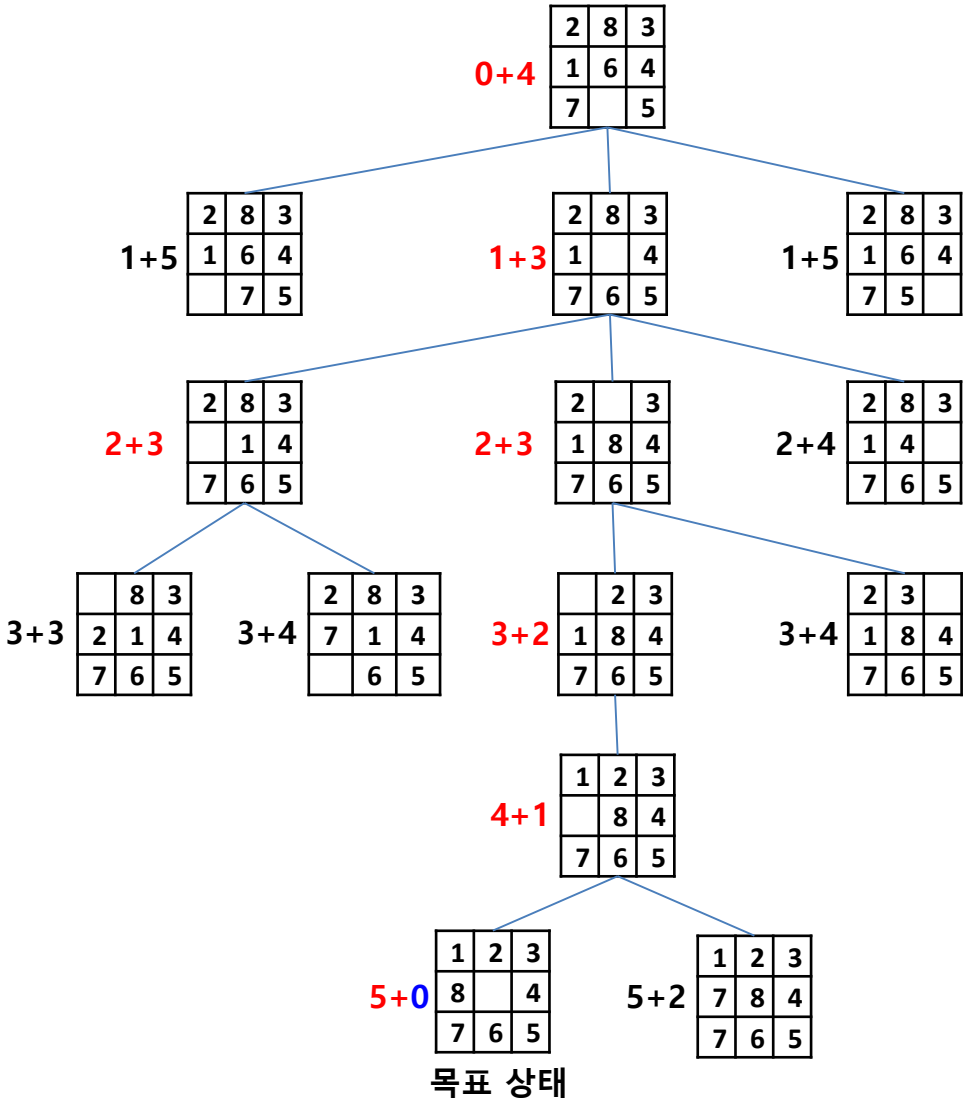
- $f(n)$: 노드 n 을 경유하는 **전체 비용**
  - 현재 노드 n까지 이미 **투입된 비용** $g(n)$과 목표 노드까지의 **남은 비용** $h(n)$의 합
  - $f(n) = g(n) + h(n)$

- $h(n)$: **남은 비용**의 정확한 예측 불가
  - $\hat{h}(n)$: $h(n)$에 대응하는 **휴리스틱 함수**(heuristic function)
- $\hat{f}(n)$: 노드 n 을 경유하는 **추정 전체 비용**
  - $\hat{f}(n) = g(n) + \hat{h}(n)$

정보이용 탐색

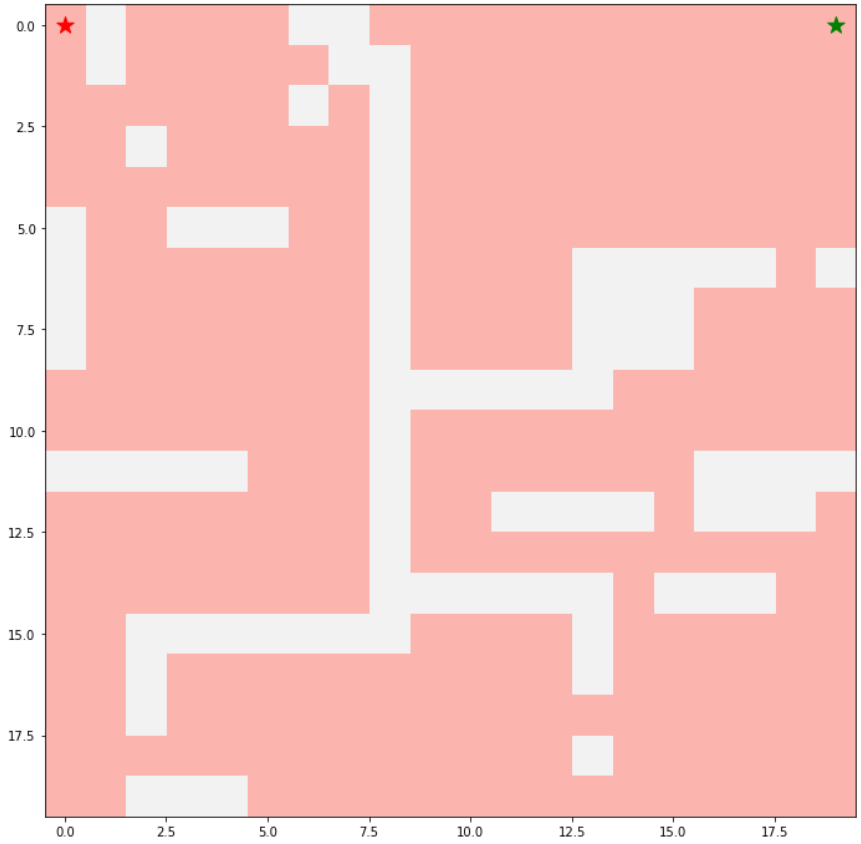
❖ 8-퍼즐 문제의 A* 알고리즘 적용

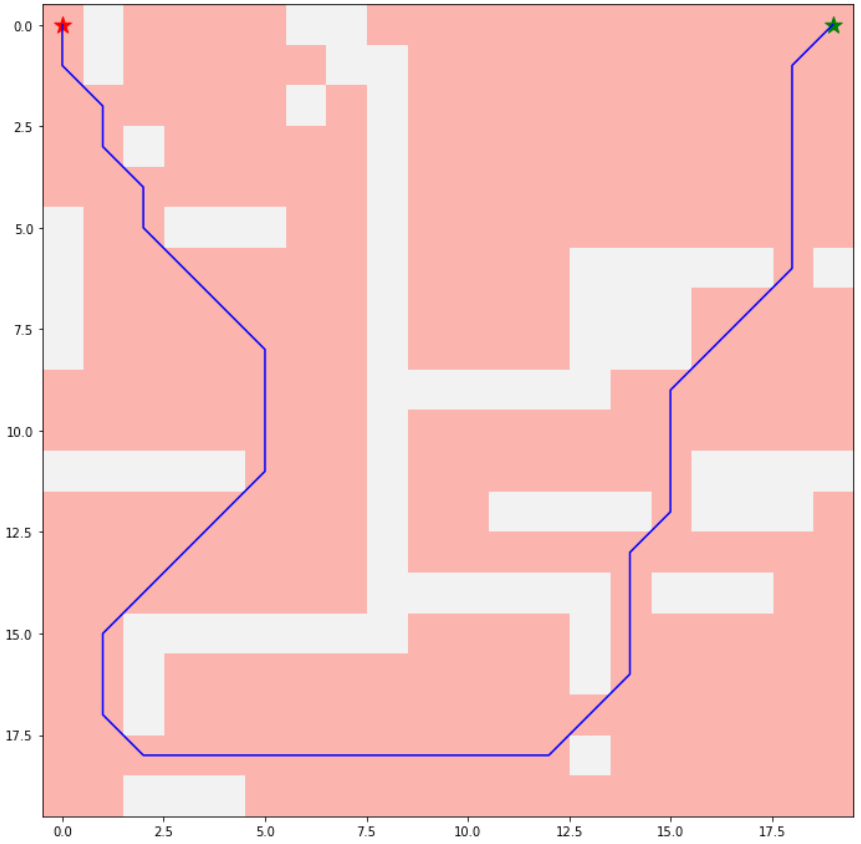
$$\hat{f}(n) = g(n) + \hat{h}(n)$$



프로그래밍 실습 : A* 알고리즘

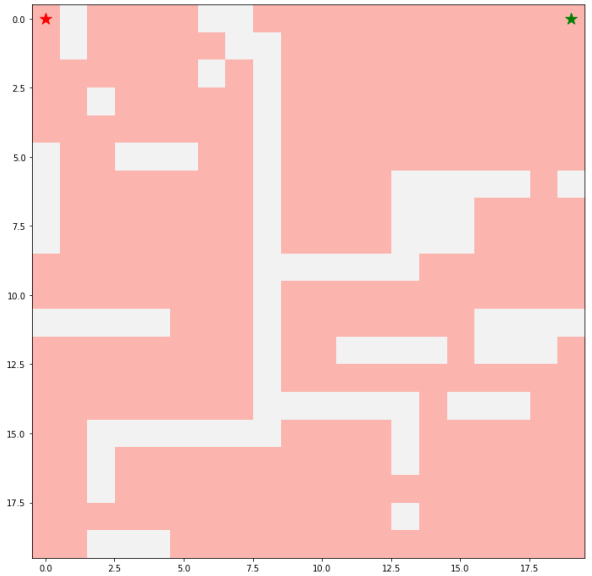
❖ 격자 공간에서 최단 경로 찾기





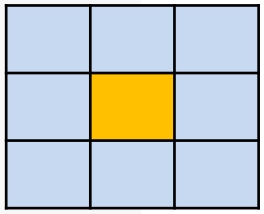
Source : www.analytics-link.com/post/2018/09/14/applying-the-a-path-finding-algorithm-in-python-part-1-2d-square-grid

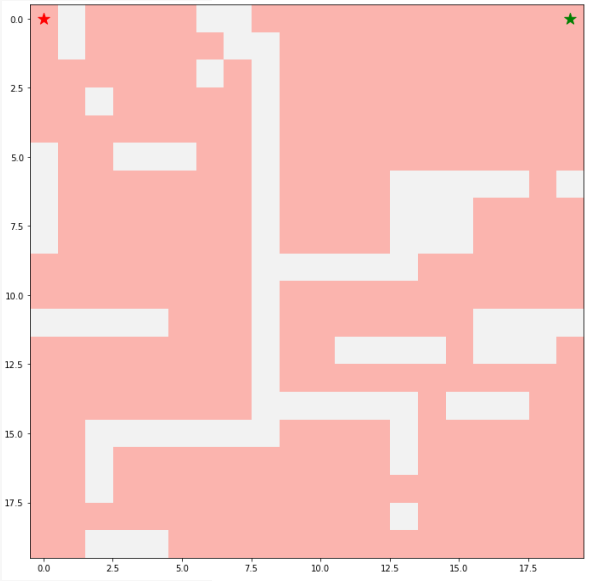
```python
import numpy as np
import heapq   # min heap을 구현하는 heap queue
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

# 지도  1:벽, 0: 빈공간
grid = np.array([
    [0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

start = (0,0)      # 시작 위치
goal = (0,19)      # 목적지 위치

# 휴리스틱 함수 h() : a와 b사이의 유클리드 거리
def heuristic(a, b):
    return np.sqrt((b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2)

```



Source : www.analytics-link.com/post/2018/09/14/applying-the-a-path-finding-algorithm-in-python-part-1-2d-square-grid

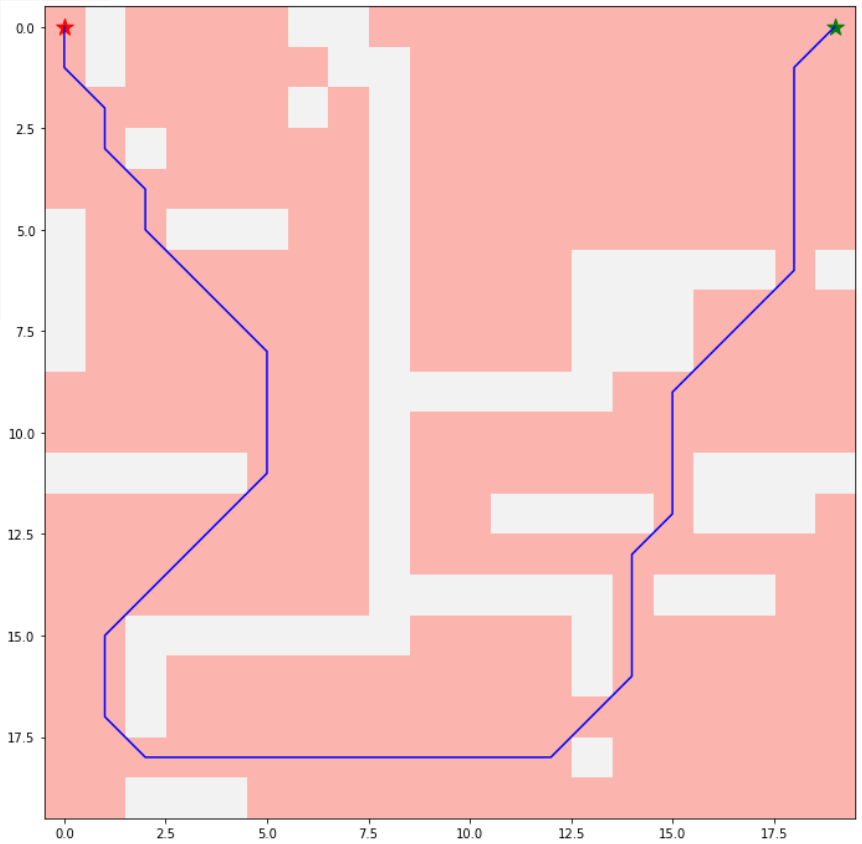
```python
# A* 알고리즘
def Astar(array, start, goal):
    neighbors = [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)] # 이웃 위치
    close_set = set()  # 탐색이 종료된 위치들의 집합
    came_from = {}
    gscore = {start:0}  # 시작 위치의 g() 값
    fscore = {start:heuristic(start, goal)}  # 시작위치의 f() 값
    oheap = []  # min-heap
    heapq.heappush(oheap, (fscore[start], start))  # (거리, 출발지) min-heap에 저장

    while oheap:
        current = heapq.heappop(oheap)[1]  # f()값이 최소인 노드 추출
        if current == goal:  # 목적지 도착
            data = []
            while current in came_from:  # 목적지에서 역순으로 경로 출출
                data.append(current)
                current = came_from[current]
            return data
        close_set.add(current)  # current 위치를 탐색이 종료된 것으로 간주

        for i, j in neighbors:  # current 위치의 각 이웃 위치에 대해 f() 값 계산
            neighbor = current[0] + i, current[1] + j   # 이웃 위치
            if 0 <= neighbor[0] < array.shape[0]:
                if 0 <= neighbor[1] < array.shape[1]:
                    if array[neighbor[0]][neighbor[1]] == 1: # 벽
                        continue
                else: # y 방향의 경계를 벗어난 상황
                    continue
            else:     # x 방향의 경계를 벗어난 상황
                continue

            temp_g_score = gscore[current] + heuristic(current, neighbor) # g^(n) = g(c)+h((c,n))
            if neighbor in close_set and temp_g_score >= gscore.get(neighbor, 0):
                continue  # 이미 방문한 위치이면서 g^()값이 기존 g()값보다 큰 경우 --> 무시

            if  temp_g_score < gscore.get(neighbor, 0) or neighbor not in [i[1]for i in oheap]:
                # g^(n) < g(n) 이거나, n을 처음 방문한 경우
                came_from[neighbor] = current  # neighbor에 도달한 최선의 경로에서 직전 위치는 current
                gscore[neighbor] = temp_g_score  # g(n) = g^(n)
                fscore[neighbor] = temp_g_score + heuristic(neighbor, goal)  # f() = g()+h()
                heapq.heappush(oheap, (fscore[neighbor], neighbor))  # min heap에 (f(), neighbor) 삽입
    return False
```





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

```
78
79 route = Astar(grid, start, goal)
80 route = route + [start] # 출발 위치 추가
81 route = route[::-1]  # 역순으로 변환
82 print('경로: ', route)
83
84 # route에서 x와 y 좌표 추출
85 x_coords = []
86 y_coords = []
87 for i in (range(0,len(route))):
88     x = route[i][0]
89     y = route[i][1]
90     x_coords.append(x)
91     y_coords.append(y)
92
93 # 지도와 경로 그리기
94 fig, ax = plt.subplots(figsize=(12,12))
95 ax.imshow(grid, cmap=plt.cm.Pastel1)
96 ax.scatter(start[1],start[0], marker = "*", color = "red", s = 200)
97 ax.scatter(goal[1],goal[0], marker = "*", color = "green", s = 200)
98 ax.plot(y_coords,x_coords, color = "blue")
99 plt.show()
```




```
경로: [(0, 0), (1, 0), (2, 1), (3, 1), (4, 2), (5, 2), (6, 3), (7, 4), (8,
5), (9, 5), (10, 5), (11, 5), (12, 4), (13, 3), (14, 2), (15, 1), (16,
1), (17, 1), (18, 2), (18, 3), (18, 4), (18, 5), (18, 6), (18, 7), (18,
8), (18, 9), (18, 10), (18, 11), (18, 12), (17, 13), (16, 14), (15, 14),
(14, 14), (13, 14), (12, 15), (11, 15), (10, 15), (9, 15), (8, 16), (7,
17), (6, 18), (5, 18), (4, 18), (3, 18), (2, 18), (1, 18), (0, 19)]
```

Quiz

❖ **맹목적 탐색 기법이 아닌 것을 다음 중에서 선택하시오.**

① 깊이 우선 탐색 ② 반복적 깊이심화 탐색 ③ A* 알고리즘 ④ 너비 우선 탐색

❖ **다음 맹목적 탐색 기법들 중에서 탐색공간이 클 때 가장 우선적으로 고려해볼 만 한 것을 선택하시오.**

① 깊이 우선 탐색 ② 반복적 깊이심화 탐색 ③ 양방향 탐색 ④ 너비 우선 탐색

❖ **탐색에 대한 설명으로 가장 적합하지 않은 것을 선택하시오.**

① 깊은 우선 탐색은 너비 우선 탐색에 비하여 메모리 사용 공간이 적다.

② 너비 우선 탐색은 최단 경로 해의 탐색을 보장하지 못한다.

③ 반복적 깊이심화 탐색은 최단 경로 해의 탐색을 보장한다.

④ 반복적 깊이심화 탐색은 깊이 우선 탐색을 반복적으로 적용하지만, 깊이 우선 탐색에 비하여 지나치게 많은 비용이 발생하지는 않는다.

Quiz

❖ **정보이용 탐색 기법이 아닌 것을 선택하시오.**

① 언덕 오르기 방법 ② 최상 우선 탐색 ③ A* 알고리즘 ④ 양방향 탐색

❖ **탐색에 대한 설명으로 가장 적합하지 않은 것을 선택하시오.**

① 최상 우선 탐색은 확장 중인 노드들 중에서 목표 노드까지 남은 거리가 가장 짧은 노드를 선택하는데, 남은 거리를 추정하기 위해 사용되는 휴리스틱에 따라 탐색 성능이 영향을 크게 받는다.

② 언덕 오르기 방법은 현재 위치에 이웃한 위치들을 평가하여 탐색을 하는 방법으로, 최적해의 탐색을 보증한다.

③ 빔 탐색은 휴리스틱에 의한 평가값이 우수한 일정 개수의 확장 가능한 노드 만을 메모리에 관리하면서 최상 우선 탐색을 적용한다.

④ A* 알고리즘은 시작노드에서 현재 노드까지 오는 데 소모된 비용과 앞으로 남은 비용에 대한 합이 최소가 되는 것을 우선적으로 탐색한다.