Heuristic Search

내용

- Heuristic 탐색
- Hill Climbing 탐색
- A* 알고리즘
- 최단 경로 탐색

Heuristic 탐색

- 문제 영역에 대한 지식을 사용할 수 있다면 탐색 작업을 훨씬 빠르게 할 수 있음
- 적용 지식은 경험 또는 직관으로부터 근거
- 이러한 방법은 경험적 탐색 또는 휴리스틱 (heuristic search) 방법임
  - TSP 문제: 현재 위치에서 가까운 도시부터 방문

8-puzzle에서의 휴리스틱

예를 들어서 현재 상태와 목표 상태가 다음과 같다고 하자

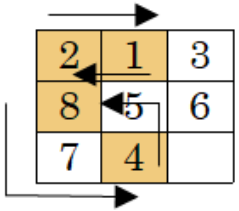
2	1	3
8	5	6
7	4	

1	2	3
4	5	6
7	8	

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

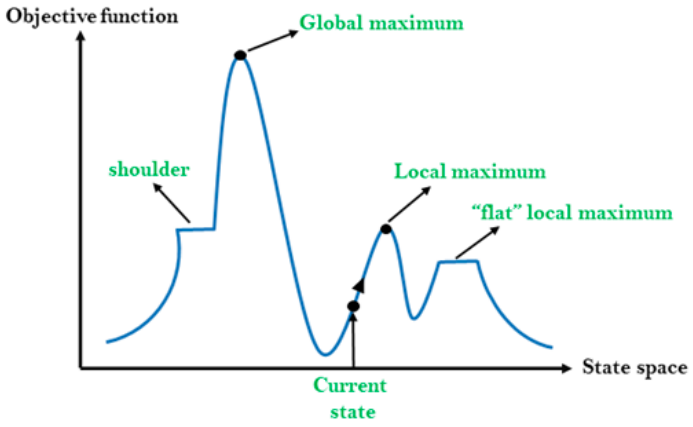
2	1	3
8	5	6
7	4	

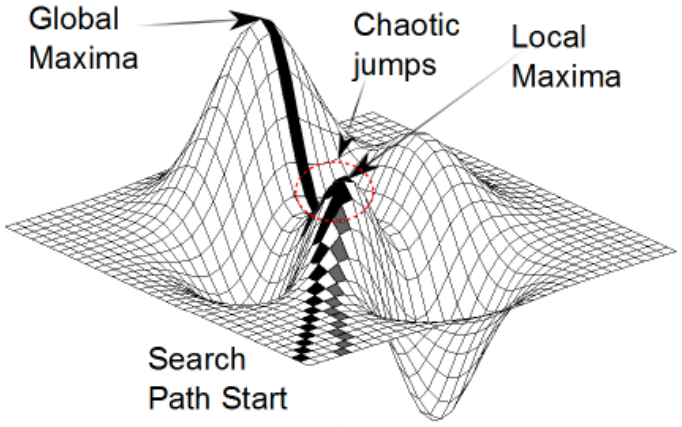
- h2(N) = 각 타일의 목표 위치까지의 거리 = 1+1+0+2+0+0+0+2 = 6



Hill Climbing

- 경험적인 탐색 방법은 무조건 휴리스틱 함수 값이 가장 좋은 노드만을 선택
- 등산할 때 무조건 현재의 위치보다 높은 위치로만 이동
- 일반적으로는 현재의 위치보다 높은 위치로 이동하면 산의 정상에 도달할 가능성이 높음





Hill Climbing

```
import random

# define function to maximize
def f(x):
    return -(x - 3)**2 + 9

def hill_climb(start, max_steps=20):
    x = start
    for step in range(max_steps):

        # neighbor is x+1 or x-1 (stay inside 0..10)
        neighbors = [n for n in (x-1, x+1) if 0 <= n <= 10]

        # pick the neighbor with the highest value
        best = max(neighbors, key=f)

        if f(best) > f(x):   # move only if it improves
            x = best
        else:
            break            # stop if no improvement
    return x, f(x)

# try from a random start
start = random.randint(0, 10)
best_x, best_val = hill_climb(start)

print(f"Start: x={start}, f(x)={f(start)}")
print(f"Best : x={best_x}, f(x)={best_val}")
```

(1) (0, 10) 범위의 임의 수 선택

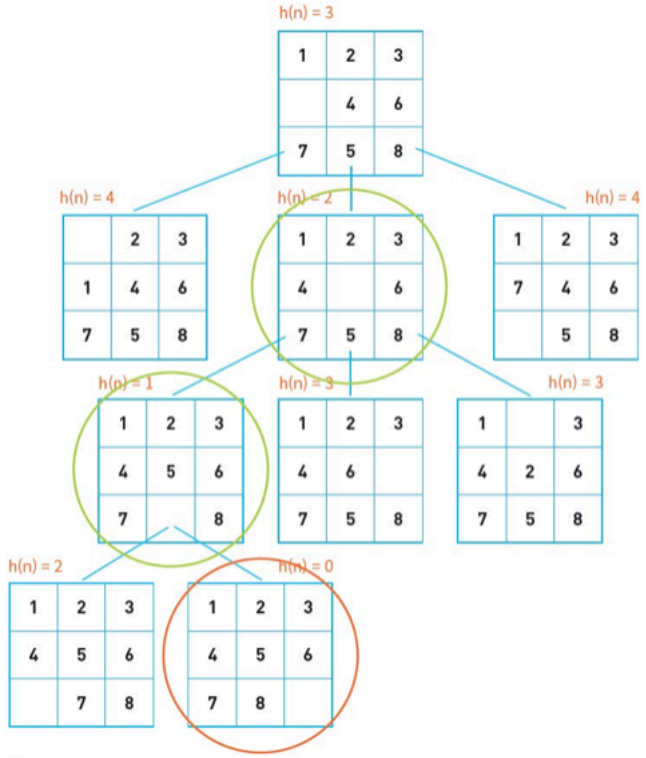
(2) 각 단계에서 두 이웃 수 x-1 와 x+1 고려

(3) 큰 값을 갖는 x 를 선택

(4) 더 큰 값을 갖는 x가 없으면 종료

Hill Climbing

- 평가 함수의 값이 좋은 노드를 먼저 선택
- 평가함수로 자신의 위치에 있지 않은 타일의 개수 사용



Hill Climbing 알고리즘

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

A* 알고리즘

- 가중치를 갖는 그래프로 부터 두 노드를 연결하는 최소 비용 경로를 찾는 기법
- A* 알고리즘은 평가 함수의 값의 정의

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

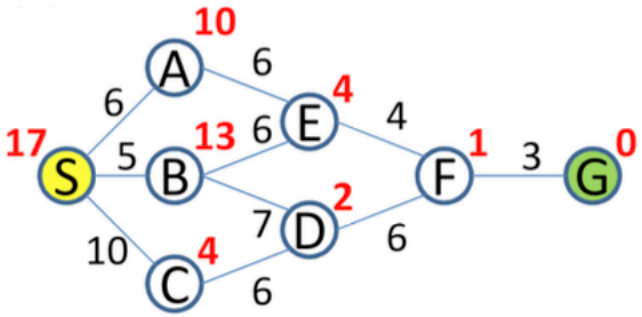
  - h(n): 현재 노드에서 목표 노드까지의 거리
  - g(n): 시작 노드에서 현재 노드까지의 비용

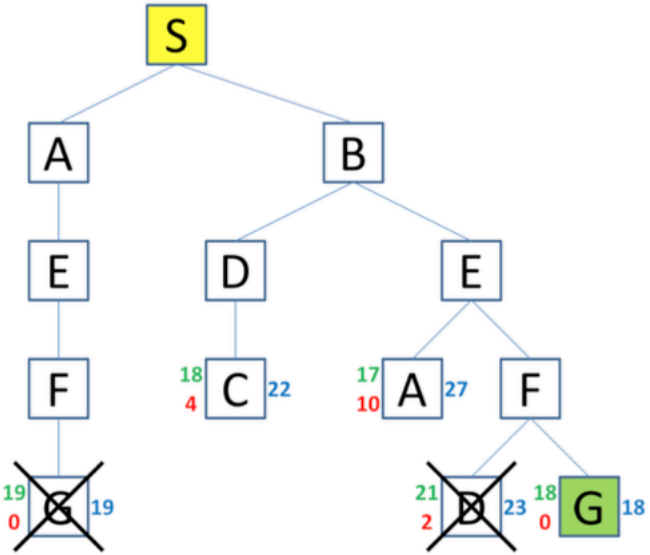
A* 알고리즘

Algorithm 2 A* algortihm

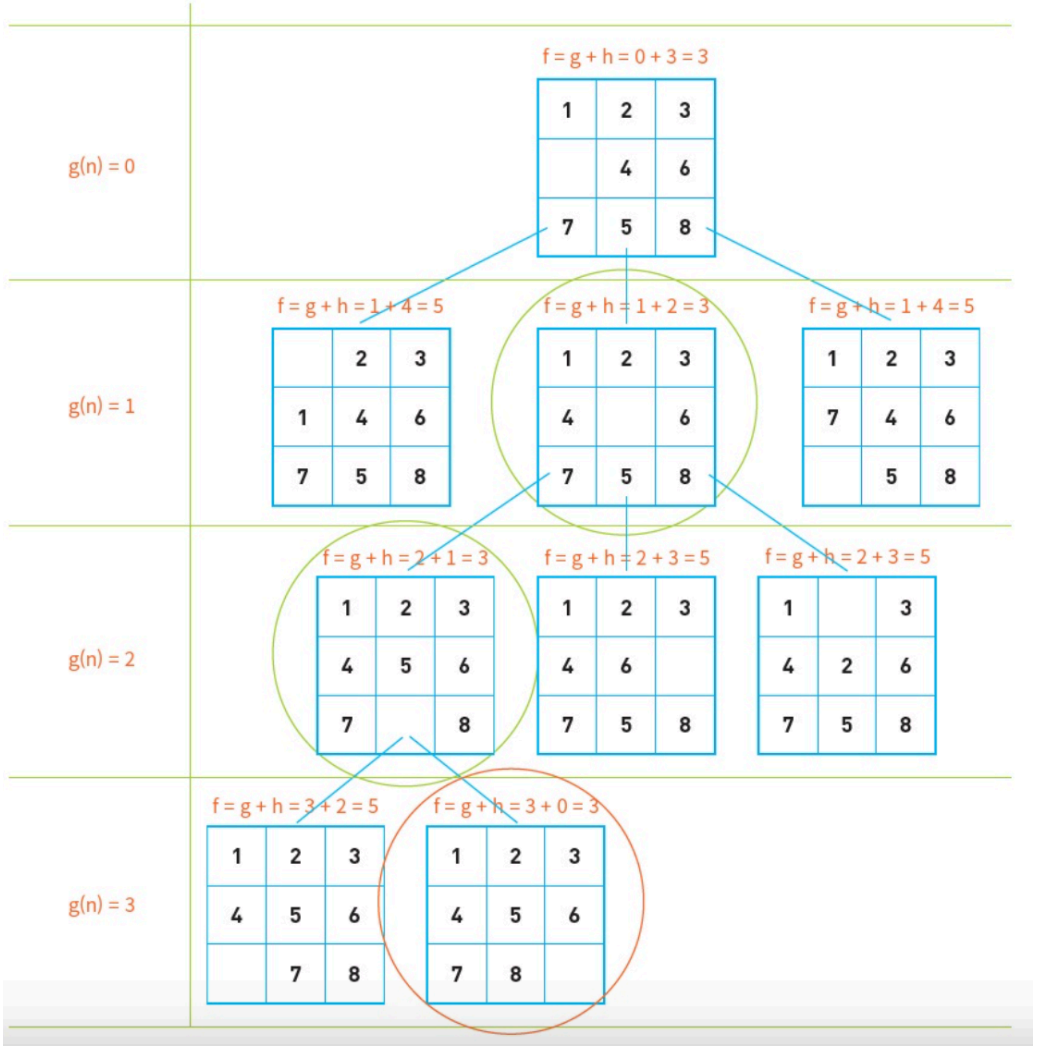
```
1: function A-Star( G, s, t )
2:     Push s into Q
3:     for all u ∈ Q do
4:         g[u] = ∞; f[u] = ∞
5:         h[u] = estimate value from u to t
6:     end for
7:     g[s] = 0; f[s] = h[s]
8:     while Q ≠ φ do
9:         u = deleteFromQ(Q, f)
10:        if u = t then
11:            return
12:        else
13:            for all v ∈ N(u) do                ▷ Find neighbors of u
14:                if v ∈ Q and (g[u] + w(u, v) < g[u]) then
15:                    g[v] = g[u] + w[u, v]
16:                    prev[v] = u
17:                    f[v] = g[v] + h[v]
18:                end if
19:            end for
20:        end if
21:    end while
22: end function
23:
24: function deleteFromQ( Q, f )
25:    Find u with the smallest f[u] return u
26: end function
```

A* 알고리즘


S
A
B
C
D
E
F
G
17
10
13
4
2
1
0
6
5
10
6
7
6
6
4
3

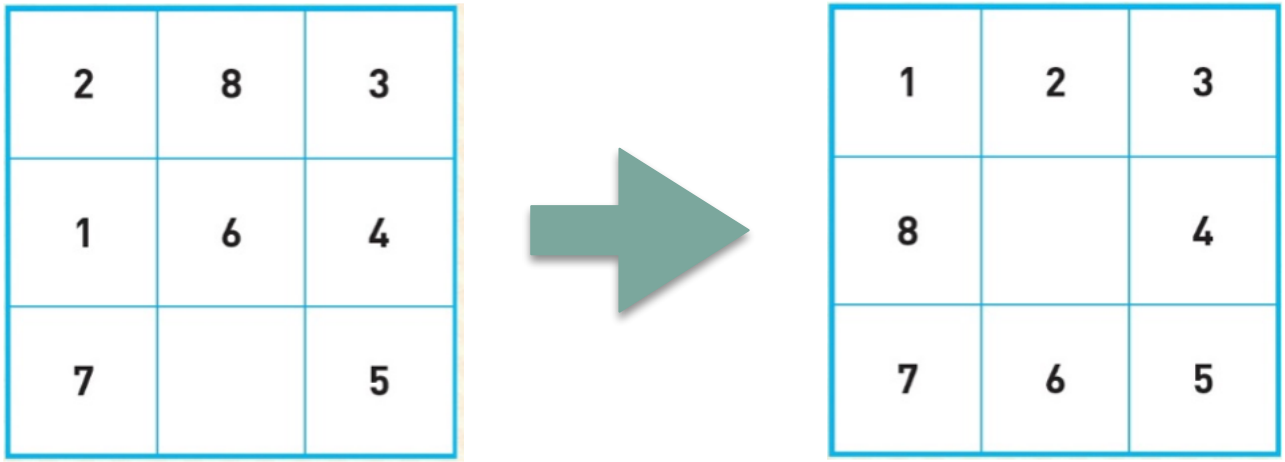

S
A
B
E
D
E
F
C
A
F
G
D
G
19
0
19
18
4
22
17
10
27
21
2
23
18
0
18

8-puzzle 에서의 A* 알고리즘

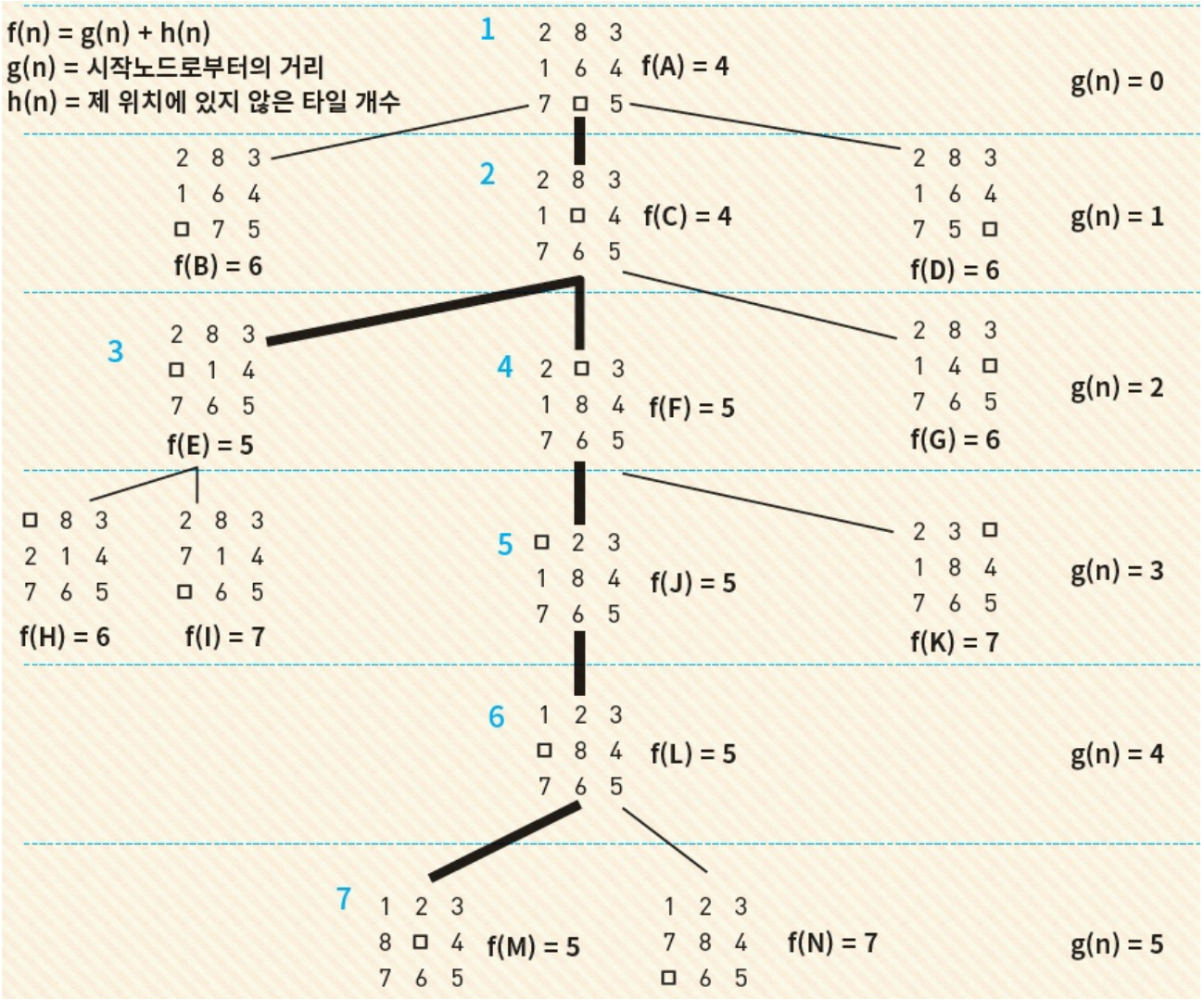


A* 알고리즘을 시뮬레이션

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



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
```

A* 알고리즘 파이썬 구현

```
  def __eq__(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(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"+\
    "-----------------"
```

A* 알고리즘 파이썬 구현

```
# 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

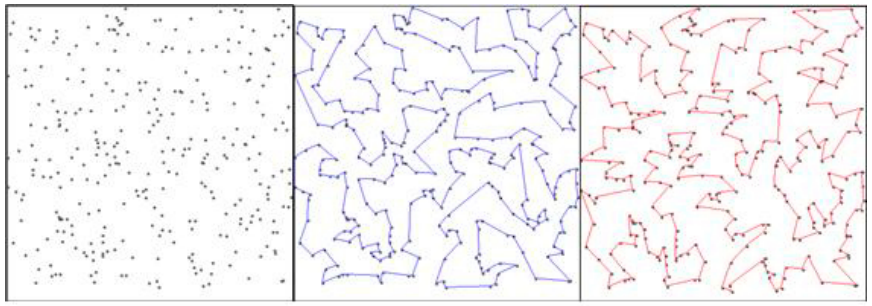
```
closed_queue = [ ]
moves = 0
while not open_queue.empty():

  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
```

TSP 문제

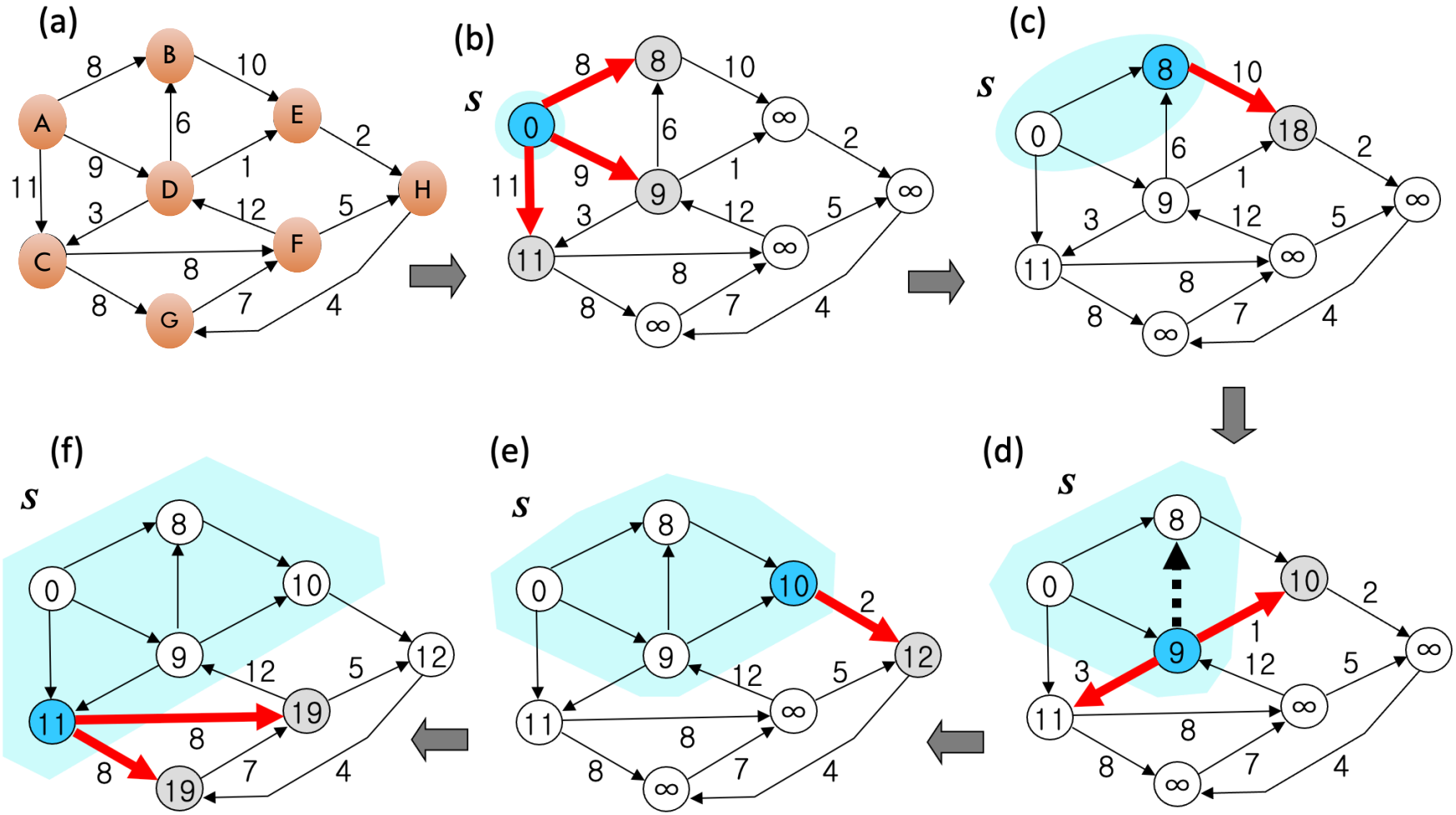
- TSP(travelling salesman problem)은 "도시의 목록과 도시들 사이의 거리가 주어졌을 때, 하나의 도시에서 출발하여 각 도시를 방문하는 최단 경로를 찾는 탐색 문제임



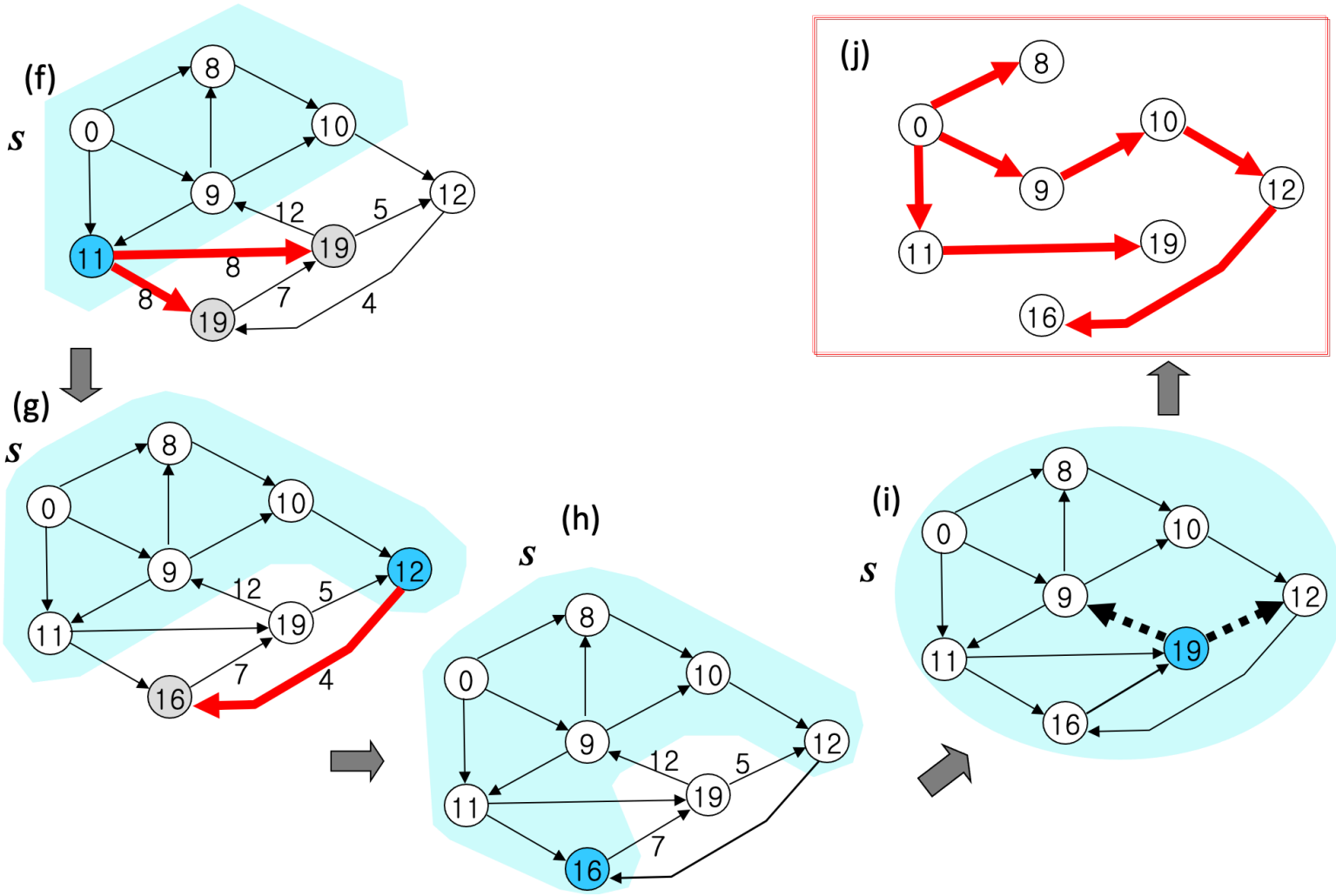
최단 경로 탐색

- 가중치가 있는 방향성을 갖는 그래프
- 방향성이 없는 그래프는 각 에지에 대해 방향성을 갖는 그래프로 고려
- 에지 (u, v)는 방향성을 갖는 (u, v)와 (v, u)를 의미 정의
- 두 노드 사이의 최단 경로
  - ✓ 두 노드 사이의 경로들 중 에지의 가중치 합이 최소인 경로
  - ✓ 간선 가중치의 합이 음인 싸이클이 있으면 문제가 정의되지 않음
- 시작 노드에서 모든 노드 간의 최단 경로
  - 시작 노드에서 모든 노드간 최단 경로

Dijkstra 알고리즘(1)



Dijkstra 알고리즘(2)



Dijkstra 알고리즘(4)

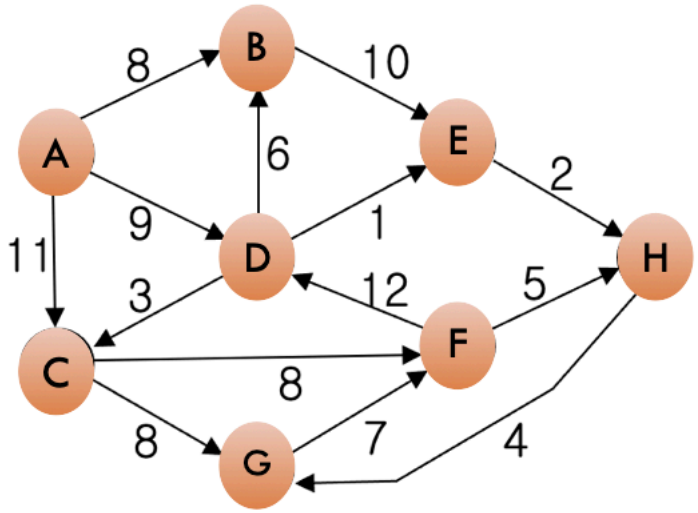
Algorithm 1 Dijkstra's Algorithm ($G = (V, E), s \in V$)

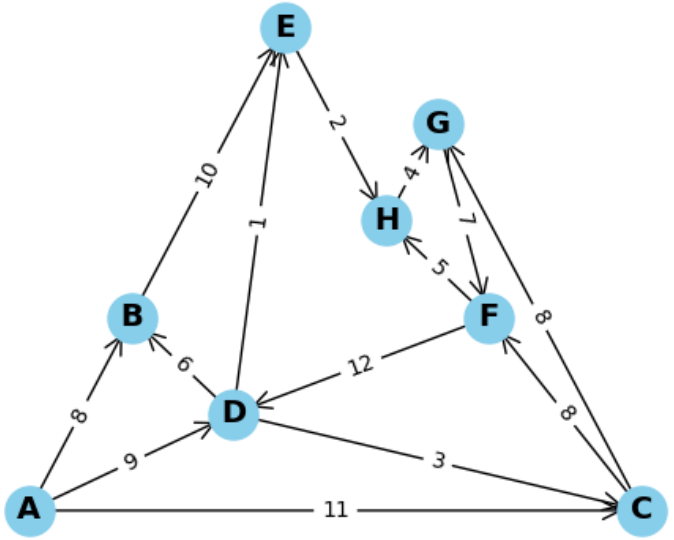
1: **Input:** Weighted directed graph $G = (V, E)$ with nonnegative weights w, source s
2: **Output:** Distances dist[v] and predecessors prev[v] for all $v \in V$
3: **Initialize:** $\forall v \in V$: dist[v] $\leftarrow \infty$; dist[s] $\leftarrow 0$; $S \leftarrow \phi; prev[v] = undefined$
4: **while** $S \neq V$ **do**
5: $u =$ EXTRACTMIN(V-S, d)
6: $S = S \cup \{u\}$
7: $\tau = \{v|(u, v) \in E\}$
8: **for all** $v \in \tau$ **do**
9: **if** dist[u] + $w(u, v)$ < dist[v] and $v \in (V - S)$ **then**
10: dist[v] = dist[u] + $w(u, v)$
11: $prev[v] = u$
12: **end if**
13: **end for**
14: **end while**
15: **return** dist

Algorithm 2 Example with a Function

1: **function** EXACTMIN($Q, d[]$)
2: **return** $u \in Q$ with the smallest distance $d[u]$
3: **end function**

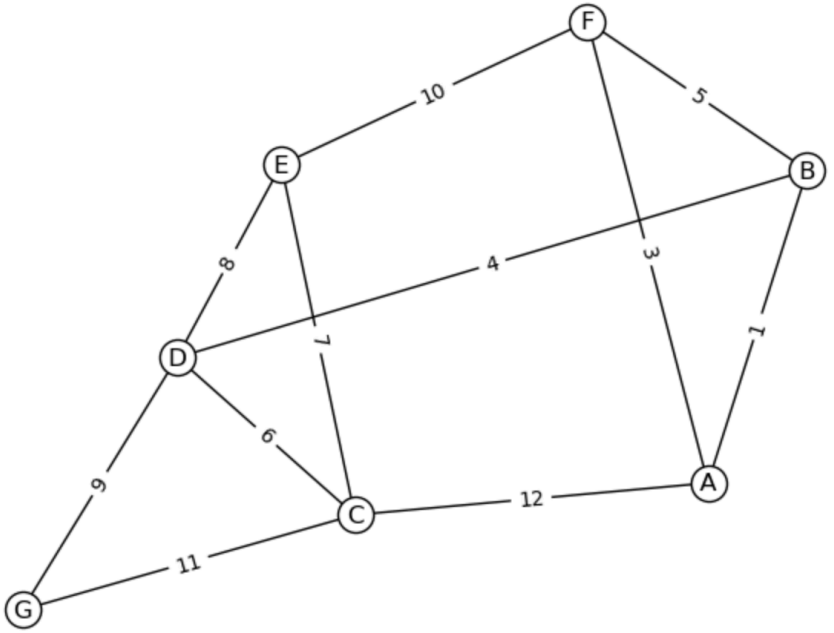
Dijkstra 알고리즘(5)






```
Distances: {'A': 0, 'B': 8, 'D': 9, 'E': 10, 'C': 11, 'H': 12, 'G': 16, 'F': 19}
Path to D: ['A', 'D']
```

Dijkstra 알고리즘(6)




```
Cost A-->G: 14 Path: ['A', 'B', 'D', 'G']

All paths:
{'A': ['A'], 'B': ['A', 'B'], 'C': ['A', 'B', 'D', 'C'], 'F': ['A', 'F'], 'D': ['A', 'B
{'B': ['B'], 'A': ['B', 'A'], 'D': ['B', 'D'], 'F': ['B', 'A', 'F'], 'C': ['B', 'D', 'C
{'C': ['C'], 'A': ['C', 'D', 'B', 'A'], 'D': ['C', 'D'], 'E': ['C', 'E'], 'G': ['C', 'G
{'F': ['F'], 'A': ['F', 'A'], 'B': ['F', 'A', 'B'], 'E': ['F', 'E'], 'C': ['F', 'A', 'B
{'D': ['D'], 'B': ['D', 'B'], 'C': ['D', 'C'], 'E': ['D', 'E'], 'G': ['D', 'G'], 'A': [
{'E': ['E'], 'C': ['E', 'C'], 'D': ['E', 'D'], 'F': ['E', 'F'], 'A': ['E', 'F', 'A'], '
{'G': ['G'], 'C': ['G', 'C'], 'D': ['G', 'D'], 'B': ['G', 'D', 'B'], 'E': ['G', 'D', 'E

From A to all nodes:
from A to [['A']]
from B to [['A', 'B']]
from C to [['A', 'B', 'D', 'C']]
from F to [['A', 'F']]
from D to [['A', 'B', 'D']]
from E to [['A', 'F', 'E'], ['A', 'B', 'D', 'E']]
from G to [['A', 'B', 'D', 'G']]
```

정리

- 탐색은 상태 공간에서 시작 상태에서 목표 상태까지의 경로를 찾는 것임
- 연산자는 하나의 상태를 다른 상태로 변경
- 맹목적인 탐색 방법(blind search method)은 목표 노드에 대한 정보를 이용하지 않고 기계적인 순서로 노드를 확장하는 방법
- 경험적 탐색 방법(heuristic search method)은 목표 노드에 대한 경험적인 정보를 사용하는 방법: Hill Climbing 기법과 A* 탐색 등
- 최단 경로 탐색 : Dijkstra 알고리즘