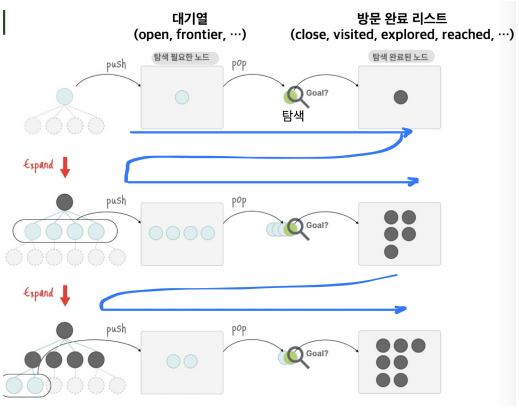
과제 0925

2315028 김성현

▼ 탐색을 통한 문제해결

- 기본 방법 (공통적인 탐색방법)



대기열 → 탐색 → 방문완료리스트 → 자식노드 확장 ...

- 대기열 추가 : 루트나 자식 노드를 대기열에 추가
  - 대기열 (=open list, frontier)
  - 탐색완료리스트 (=close, visited, explored, reached) ← 이미 갔던 경로는 다시 가지 못하도록 함
- 탐색 : 대기열에서 노들르 꺼내 목표노드와 비교
- 확장 : 자식 노드를 생성

- 종류

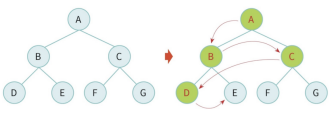
▼ 정보 없는 탐색

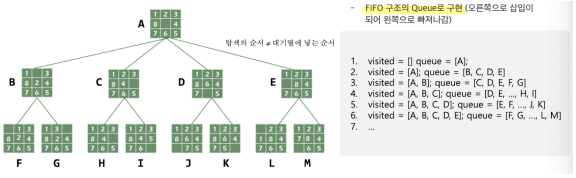
: 아무런 정보 없이 탐색하는 방법. (=맹목적인 탐색) 기계적인 순서로 노드를 확장 및 탐색하는 것으로 매우 소모적인 탐색

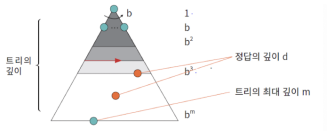
▼ 너비 우선 탐색 (BFS, breath-first search)

: 너비(폭)를 우선적으로 탐색하는 방법. 모든 동작의 비용이 동일할때 적합한 방법

FIFO구조의 Queue로 구현(오른쪽 → 왼쪽). 탐색의 순서 ≠ 대기열에 넣는 순서








```
Function BFS(initial_state, goal_state)
  queue ← [initial_state]
  visited ← [ ]
  while queue != [ ] do
    current_state ← queue의 첫번째 요소
    if current_state == goal_state
      return SUCCESS
    else
      current_state를 visited에 추가
      current_state의 자식 노드를 생성
      if current_state의 자식 노드가 이미 queue나 visited에 있다면
        해당 자식 노드 건너뜀
      else
        남은 자식 노드들은 queue의 마지막에 추가
  return FAIL
```

의사코드

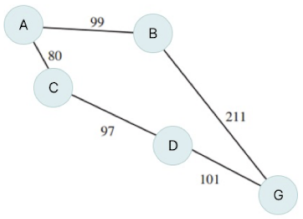
- 완결성 : 분기계수가 유한하면 반드시 해답을 발견할 수 있음. 무한하면 못 찾을 수 있음
- 최적성 : 가장 경로가 짧은 최적의 해답을 찾을 수 있음(모든 경로를 다 보기에)
- 시간 및 공간 복잡도 : O(b^d)-지수복잡도 ← 깊이에 따라 계속 증가함 (단점)

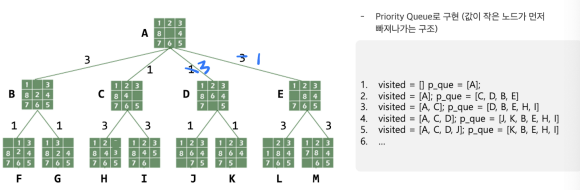
▼ 균일 비용 탐색 (UCS, uniform-cost search)

: (=다익스트라 알고리즘), 가장 적은(누적경로)비용으로 목표상태에 도달하는 경로를 찾는 방법

모든 동작의 비용이 동일할 시 BFS와 동일하게 동작

Priority Queue로 구현 (값이 작은 노드가 먼저 빠져나가는 구조)





up, down = 3 / left, right = 1

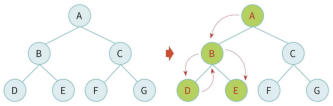
```
Function UCS(initial_state, goal_state)
  p_queue ← [(cost, initial_state)]
  visited ← [ ]
  while p_queue != [ ] do
    current_cost, current_state ← p_queue 에서 경로 비용이 가장 낮은 상태
    if current_state == goal_state
      return SUCCESS
    else
      current_state를 visited에 추가
      current_state의 자식 노드 생성
      if current_state의 자식 노드가 p_queue 이나 visited 에 있으면
        해당 자식 노드 건너뜀
      else
        자식 노드의 경로 비용을 계산
        자식 노드들을 p_queue에 추가
  return FAIL
```

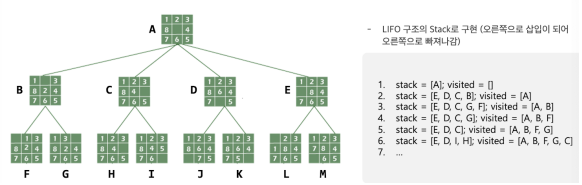
의사코드

▼ 깊이 우선 탐색

: 더이상 자식노드가 없을때까지 계속 밑으로 탐색하는 기법

LIFO구조의 stack으로 구현 (오른쪽으로 삽입, 삭제)

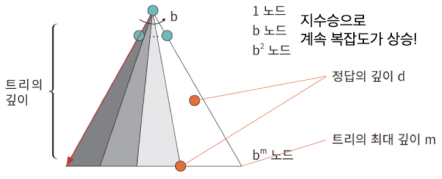




넣는 과정에서 순서를 뒤집는 것을 해줘야됨!
(reverse함수 사용)

```
1 Function DFS(initial_state, goal_state)
2   stack ← [initial_state]
3   visited ← [ ]
4   while stack != [ ] do
5     current_state ← stack의 첫번째 요소
6     if current_state == goal_state        ← reverse 함수사용
7       return SUCCESS
8     else
9       current_state를 visited에 추가
10      current_state의 자식 노드 생성
11      if current_state의 자식 노드가 이미 stack이나 visited에 있다면
12        해당 자식 노드 건너뜀
13      else
14        남은 자식 노드들은 stack의 처음에 추가
15  return FAIL
```

의사코드



탐색하지 못하는 공간이 나타날 수 있음!

(최적의 경로를 찾지 못하고 끝날 경우가 다분)

특징)

- 완결성 : 무한한상태에서는 완결적이지 않음

-시간복잡도 : O(b^m) 지수 승 만큼의 복잡도 지님. 깊이에 따라 달라짐

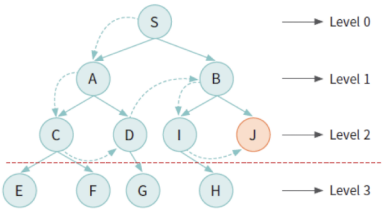
-공간복잡도 : O(bm) 메모리에 모든 상태를 담고있을 필요 없음

-최적성 : 가장 경로가 짧은 최적의 해답을 발견할 수 없음. 가장 왼쪽에 있는 해답을 발견

▼ 깊이 제한 탐색

: 깊이를 제한해 그 깊이 이상은 탐색하지 않는 방법

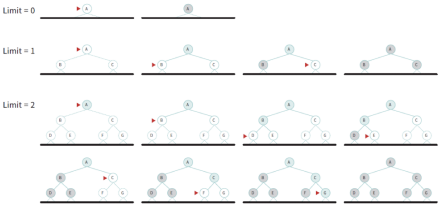
특징) 메모리 소모가 적음. 정답을 빠르게 찾을 가능성 존재. 단, 한번 잘못된 경로로 빠지면 나오지 못할 수 있음




```
Function DLS(initial_state, goal_state)
  stack ← [initial_state]
  visited ← [ ]
  while stack != [ ] do
    current_state ← stack의 첫번째 요소
    if current_state == goal_state
      return SUCCESS
    if 현재 깊이 > 깊이 제한
      다음 반복으로 건너뛰기
    else
      current_state를 visited에 추가
      current_state의 자식 노드 생성
      if current_state의 자식 노드가 이미 stack이나 visited에 있다면
        해당 자식 노드 건너뜀
      else
        남은 자식 노드들은 stack의 처음에 추가
  return FAIL
```

▼ 반복 심화 탐색

: 깊이 제한 탐색을 활용해, 한계깊이를 늘려가며 목표를 찾을때까지 탐색



특징) 공간효율성(ft.깊이우선탐색) + 완전성(ft.너비우선탐색)

해답이 존재하면 가장 적은 비용을 갖는 경로를 찾음

여러번 동일한 노드를 방문하는것이 그렇게 비용이 많이 들진 않음

▼ 정보 있는 탐색

: 목표 노드를 향한 특별한 방향성 없이 모든 상태공간을 탐색하고자 함. 맹목적 탐색. 휴리스틱이라는 힌트를 사용! (=경험적탐색방법)

휴리스틱 : 경험적인 정보. 실제 비용에대한 어림값. 정의하는 방법이 따로 없음 ex_경로이동문제 - 도시간 실제 거리가 아닌 휴리스틱으로 간단히해 정보 이용

- 최우선 탐색
- A* 탐색

▼ 8puzzle 탐색버전

```python
# 클래스 설계 (상태, 전이모형, 그외 함수 포함)
# 깊이 정보와 목표 상태 및 경로 비용 함수 추가
중단점을 추가하려면 클릭합니다.
class Puzzle:
  def __init__(self, board, depth=0):
    self.board = board # 현재 상태
    self.depth = depth # 현재 깊이


  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 Puzzle(new_board, depth) # 목표 및 깊이 정보 추가 & 경로 비용 업데이트


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

  def cost(self):
    return self.depth

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

  def __eq__(self, other):
    return self.board == other.board

  def __ne__(self, other):
    return self.board != other.board
```

```python
def cost(self):
  return self.depth

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

def __eq__(self, other):
  return self.board == other.board

def __ne__(self, other):
  return self.board != other.board

# 우선순위 큐 사용 시 값의 비교가 발생함.
# 이 때문에 less/greater than 연산을 사전 정의해야 오류 발생 안함
def __lt__(self, other):
  return self.cost() < other.cost()

def __gt__(self, other):
  return self.cost() > other.cost()
```

```python
class PuzzleC:
    def __init__(self, board, depth=0, cost=0):
        self.board = board
        self.depth = depth
        self.path_cost = cost

    def get_new_board(self, i1, i2, depth, cost):
        new_board = self.board[:]
        new_board[i1], new_board[i2] = new_board[i2], new_board[i1]
        return Puzzle(new_board, depth, cost)

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

    def cost(self):
        return self.depth

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

    def __eq__(self, other):
        return self.board == other.board

    def __ne__(self, other):
        return self.board != other.board
```

```python
  def cost(self):
      return self.depth

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

  def __eq__(self, other):
      return self.board == other.board

  def __ne__(self, other):
      return self.board != other.board

# 우선순위 큐 사용 시 값의 비교가 발생함.
# 이 때문에 less/greater than 연산을 사전 정의해야 오류 발생 안함
  def __lt__(self, other):
      return self.cost() < other.cost()

  def __gt__(self, other):
      return self.cost() > other.cost()
```

균일 비용 알고리즘을 위한 퍼즐 업데이트 버전

- self.depth : 깊이 정보
- self.goal : 목표상태
- self.path_cost : 경로비용
- expand() : 함수 내 경로비용 업데이트. 경로비용이 계속해서 누적됨
  - 위, 아래 = 3

- 왼쪽, 오른쪽 = 1

▼ 너비우선탐색

: 루트노드의 모든 자식노드들을 탐색한 후에 해가 발견되지 않으면 한 레벨 내려가서 동일한 방법으로 탐색을 계속해서 이어나가는 방법. FIFO, Queue를 활용

의사코드

```
Function BFS(initial_state, goal_state)
  queue ← [initial_state]
  visited ← [ ]
  while queue != [ ] do
    current_state ← queue의 첫번째 요소
    if current_state == goal_state
      return SUCCESS
    else
      current_state를 visited에 추가
      current_state의 자식 노드를 생성
      if current_state의 자식 노드가 이미 queue나 visited에 있다면
        해당 자식 노드 건너뜀
      else
        남은 자식 노드들은 queue의 마지막에 추가
  return FAIL
```

```
Function BFS(initial_state,
  queue ← [initial_state]
  visited ← [ ]
  while queue != [ ] do
    current_state ← queue의
    if current_state == goa
      return SUCCESS
    else
      current_state를 visit
      current_state의 자식 노
      if current_state의 자
        해당 자식 노드 건너뜀
      else
        남은 자식 노드들은 que
  return FAIL
```

```python
from collections import deque

def run_bfs(initial_state, goal_state):
    queue = deque()  # FIFO 구조의 큐 사용
    visited = []

    queue.append(initial_state)

    count = 1

    while queue:
        current_state = queue.popleft()  # FIFO 구조의 큐에서 가장 처음에 들어간 왼쪽 요소부터 꺼내기
        print(f"Count:{count}, Depth:{current_state.depth}\n{current_state}")
        count += 1

        if current_state.board == goal_state:
            return "탐색 성공"

        depth = current_state.depth + 1
        visited.append(current_state)

        for state in current_state.expand(depth):
            if (state in visited) or (state in queue):
                continue
            else:
                queue.append(state)
    # 큐가 비면 탐색 실패
    return "탐색 실패"

initial_state = PuzzleC(start)
answer = run_bfs(initial_state, goal)
print(answer)
```

```
count:35, depth:5
[8, 6, 3]
[2, 0, 4]
[1, 7, 5]
------------------
success
```

35번만에 탐색 성공

깊이 : 5

▼ 균일비용탐색

: 가장 적은 비용으로 목표 상태에 도달하는 경로를 찾는데 사용하는 방법

priority Queue(우선순위 큐) 대기열을 활용하며 각 노드는 튜플을 활용해 경로비용, 상태로 표현

의사코드

```
Function UCS(initial_state, goal_state)
  p_queue ← [(cost, initial_state)]
  visited ← [ ]
  while p_queue != [ ] do
    current_cost, current_state ← p_queue 에서 경로 비용이 가장 낮은 상태
    if current_state == goal_state
      return SUCCESS
    else
      current_state를 visited에 추가
      current_state의 자식 노드 생성
      if current_state의 자식 노드가 p_queue 이나 visited 에 있으면
        해당 자식 노드 건너뜀
      else
        자식 노드의 경로 비용을 계산
        자식 노드들을 p_queue에 추가
return FAIL
```

```
Function UCS(initial_state,
  p_queue ← [(cost, initial
  visited ← [ ]
  while p_queue != [ ] do
    current_cost, current_s
    if current_state == goa
      return SUCCESS
    else
      current_state를 visit
      current_state의 자식 노
      if current_state의 자식
        해당 자식 노드 건너뜀
      else
        자식 노드의 경로 비용을
        자식 노드들을 p_queue
return FAIL
```

```python
import heapq

def run_ucs(initial_state, goal_state):
    # 실습
    pqueue = []
    visited = []
    heapq.heappush(pqueue, (initial_state.cost(), initial_state))

    count =1
    while pqueue:
        current_cost, current_state = heapq.heappop(pqueue)
        print(f"count:{count}, depth:{current_state.depth} \n{current_state}")
        count = 1

        if current_state.board == goal_state:
            return "success"

        depth = current_state.depth + 1
        visited.append(current_state)
        for state in current_state.expand(depth):
            if (state in visited) and (state in [s[1] for s in pqueue]):
                continue
            else:
                heapq.heappush(pqueue, (state.cost(), state))

# 초기 상태와 목표 상태는 Puzzle 클래스로 정의된다고 가정
initial_state = Puzzle(start)
answer = run_ucs(initial_state, goal)
print(answer)
```

```
Count:61, Depth:5, Cost: 11
[8, 6, 3]
[2, 0, 4]
[1, 7, 5]
------------------
탐색 성공
```

61번만에 탐색 성공

비용 : 11 / 깊이 : 5

▼ 깊이우선탐색, 깊이제한탐색

: 노드들이 더이상 자식노드가 없는 수준까지 나아간 후, 이전노드로 후퇴하며 탐색하는 기법

탐색 효율을 고려해 깊이 제한 방식으로 구현. LIFO 대기열 활용. stack

```
Function DLS(initial_state, goal_state)
  stack ← [initial_state]
  visited ← [ ]
  while stack != [ ] do
    current_state ← stack의 첫번째 요소
    if current_state == goal_state
      return SUCCESS
    if 현재 깊이 > 깊이 제한
      다음 반복으로 건너뛰기
    else
      current_state를 visited에 추가
      current_state의 자식 노드 생성
      if current_state의 자식 노드가 이미 stack이나 visited에 있다면
        해당 자식 노드 건너뜀
      else
        남은 자식 노드들은 stack의 처음에 추가
  return FAIL
```

```
Function DLS(initial_state,
  stack ← [initial_state]
  visited ← [ ]
  while stack != [ ] do
    current_state ← stack의
    if current_state == goa
      return SUCCESS
    if 현재 깊이 > 깊이 제한
      다음 반복으로 건너뛰기
    else
      current_state를 visit
      current_state의 자식 노
      if current_state의 자식
        해당 자식 노드 건너뜀
      else
        남은 자식 노드들은 sta
  return FAIL
```

```python
from collections import deque

def run_dfs(initial_state, goal_state):
    # 실습
    stack = deque()
    visited = [ ]

    stack.append(initial_state)
    count = 1

    while stack:
        current_state = stack.pop()
        print(f"count:{count}, depth:{current_state.depth}\n{current_state}")
        count += 1

        if current_state.board == goal_state:
            return "success"

        depth = current_state.depth + 1
        visited.append(current_state)

        if depth > 5:
            continue

        for state in reversed(current_state.expand(depth)):
            if(state in visited) or (state in stack):
                continue
            else:
                stack.append(state)

    return "failed"

initial_state = Puzzle(start)
answer=run_dfs(initial_state, goal)
print(answer)
```

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

7번만에 탐색 성공

깊이 : 5