인공지능

Problem Solving Agents

시스템경영공학부 이지환 교수

인공지능이란?

- 인공지능 (Artificial Intelligence)
- 인공지능을 바라보는 네 가지 관점

Act? Thinks? \ Rationally? Humanly?	
System that thinks like humans	System that thinks rationally
System that acts like human	System that act rationally

합리적 Agent

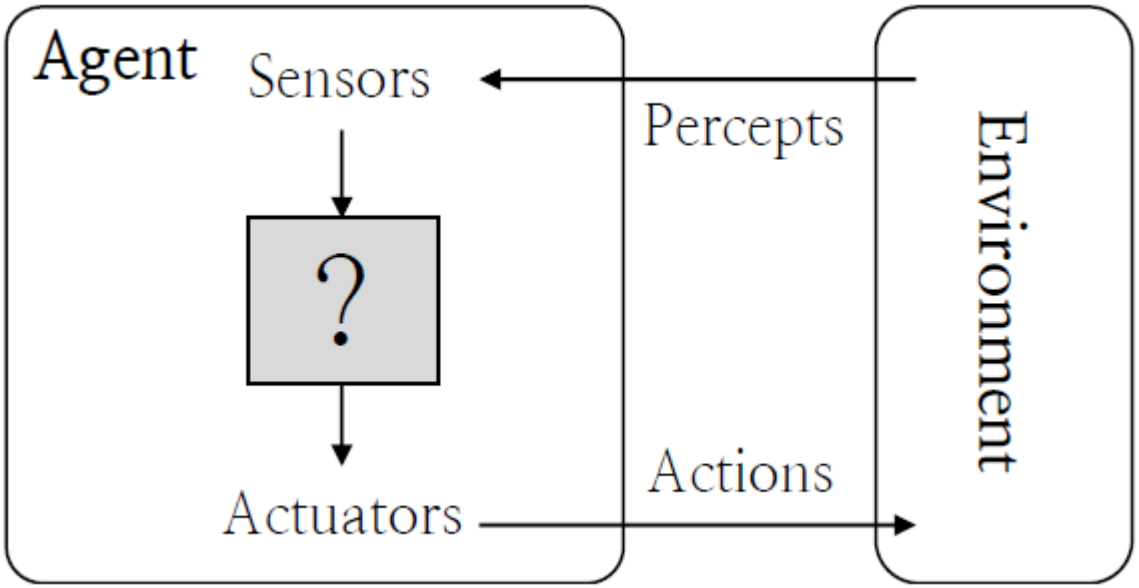
- Agent: 행동의 주체

- 합리적 (rational) Agent
  - 인공지능의 목표
  - 주어진 정보하에서 최선의 결과를 얻도록 행동하는 컴퓨터 프로그램/machine을 만드는 것

- 합리적 Agent의 특징
  - 자동적인 제어하에서 움직인다.
  - 주어진 환경을 인식한다
  - 변화에 적응한다.
  - 목적을 실현할 수 있는 능력이 있다.

합리적 Agent의 구성요소

- Sensor(감각기)를 통해 환경(enviorenment)을 인식(percept)
- Actuator(구동기)를 통해 환경에 영향을 주는 행동을 취한다. (action)



로봇청소기 agent

예시) 로봇 청소기 agent

Function

Percept	Action
먼지가 있다	빨아들인다
먼지가 없다	앞으로 이동

Program

```
function action=vacuum(state)
  if state = '먼지가 있다'
    then action = '빨아들인다'
  elseif state = '먼지가 없다'
    then action = '앞으로 이동'
```

문제해결 Agent (Problem-solving agent)

- 목적(goal)을 달성하기 위한 일련의 행위(action)의 순서(sequence)를 탐색하는 agent

- 현재상태(state)를 인식(percept)하여 목적상태(state)로 가기위한 행동(action)을 찾는다.

- 예시) 강남에서 종로로 길을 찾아가는 agent
  - 시작상태: 강남에 위치
  - 목표: 종로에 위치
  - 가능한 행동
    - 시계방향으로 1도 틀기
    - 한걸음 걷기
    - 삼각김밥 사먹기 (적절한 수준으로 행동을 정의할 필요 있음)
    - 다른 장소로 이동하기

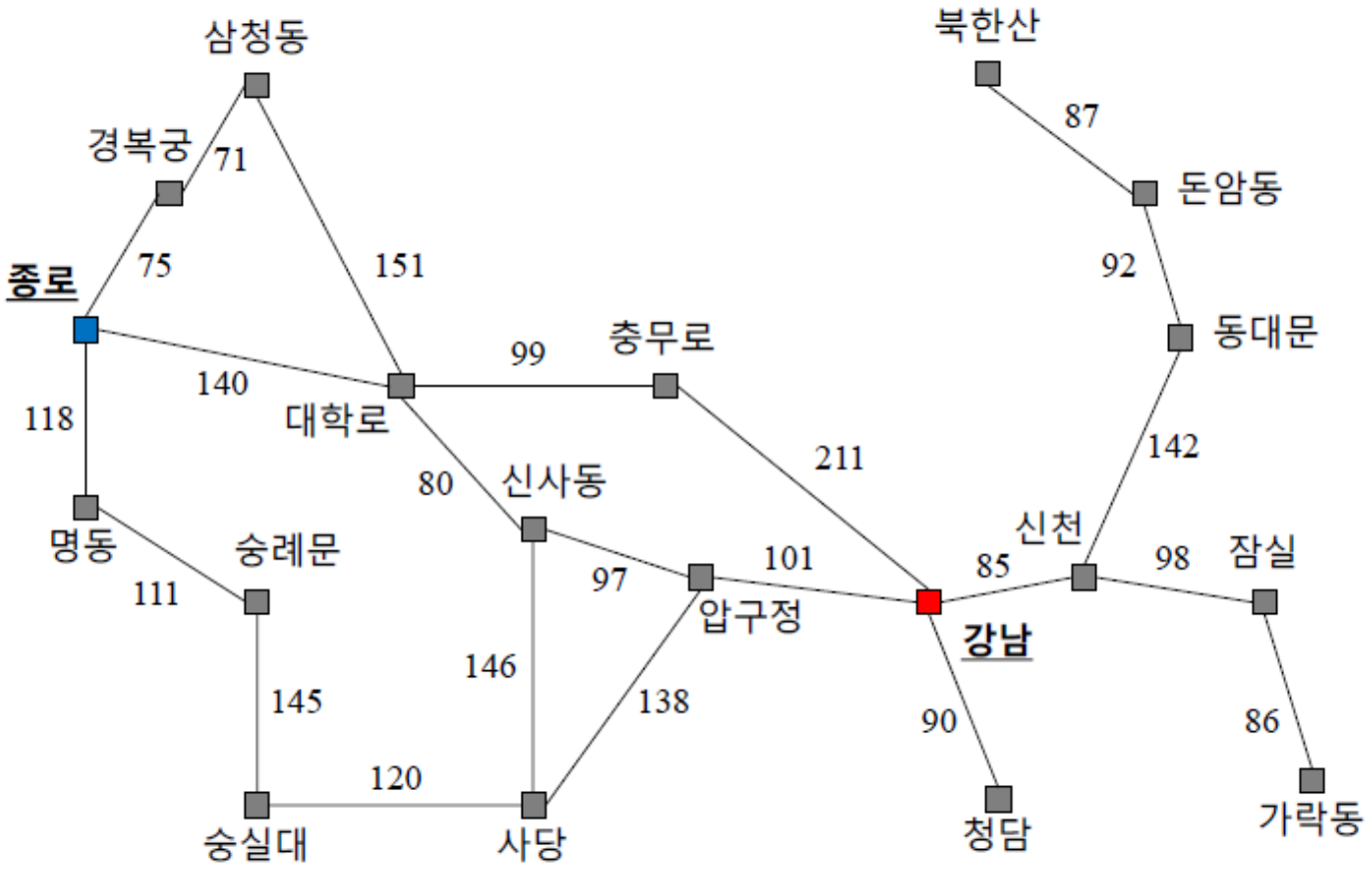
문제해결 agent

- 예시) 강남에서 종로로 길을 찾아가는 agent
  - 시작상태: 강남에 위치
  - 목표: 종로에 위치
  - 가능한 행동: 시계방향으로 1도 틀기, 한걸음 걷기, 삼각김밥 사먹기, 다른 도시로 이동하기

- 탐색(search)
  - Agent가 목적하는 상태로 가기 위해서는
  - 내가 취할 수 있는 다양한 행위 중에 가장 우수한 행위를 골라 무엇을 할지 결정하는 과정이 필요하다.
  - 이처럼 행위의 순서를 찾아나가는 과정을 search(탐색)이라고 한다.

서울의 관광명소 지도

- 시작상태: '종로'
- 목표상태: '강남'에 가기



문제의 정의 (Problem Formulation)

(1) State space: 문제와 관련된 agent의 상태. 예를 들어, 여행 경로 찾기 문제에서는 agent의 현재 위치가 state가 되고, agent가 갈 수 있는 명소들이 agent가 가질 수 있는 state의 집합, 즉 state space가 된다.

(2) Initial state: 문제를 풀기 전 최초 시점에서 agent가 가진 상태. Agent가 종로에서 여행을 시작한다면 initial state는 'In(종로)'가 된다.

(3) Actions: Agent가 취할 수 있는 action. 일반적으로 action과 action의 결과로 얻어진 state를 하나의 쌍으로 묶어 정의한다. 예를 들어
{〈Go(대학로), In(대학로)〉, 〈Go(경복궁), In(경복궁)〉,〈Go(명동), In(명동)〉}이 'In(종로)' 상태에서 취할 수 있는 action의 집합이다.

(4) Goal test: 현재 상태가 목적을 만족시켰는지 판단하는 방법. 즉, goal test를 정의하는 것은 goal을 정의하는 것과 같다. 예시에서는 '강남'에 가는 것이 목적이므로 현재 state가 'In(강남)'인지를 체크하면 된다.

(5) Path cost: 일련의 action들이 이루는 path에 수치적인 비용을 할당하는 함수. 예를 들어 명소들을 지나치면서 이동한 거리를 path cost로 삼을 수 있다. 하나 하나의 action에 대해 발생하는 비용은 step cost라 한다. Path를 구성하는 action들의 step cost를 더하면 path cost가 된다.

실습_1: 지도의 표현

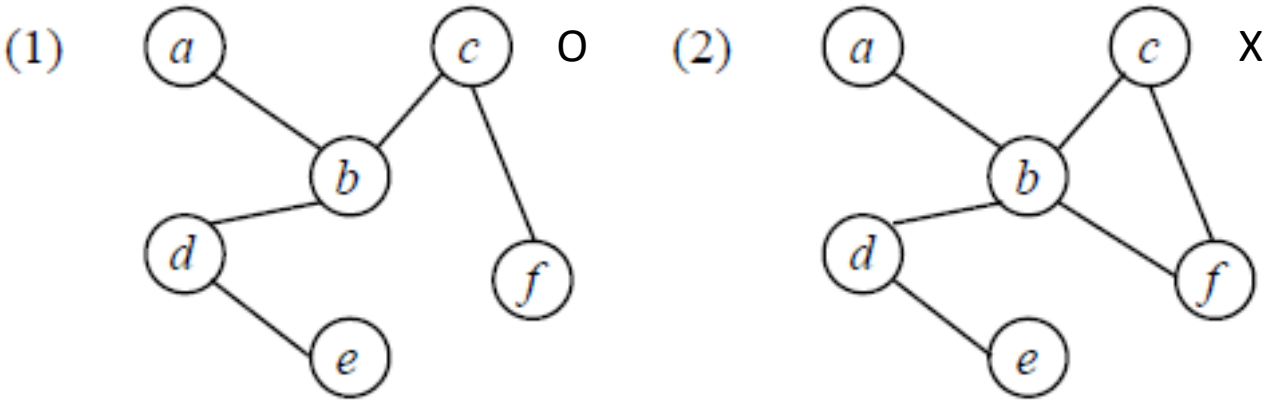
```python
# 여행자가 도달할 수 있는 state
route_states=['JR','KBG','SCD','DHR','CMR','GN','SC',
'DDM','DAD','BHS','MD','SRM','SSU','SD','SSD','AGJ','CD','JS','GRD']

import collections
route_actions=collections.OrderedDict()

# 각 state에서 취할 수 있는 행동과 cost
route_actions = {'JR':{'KBG':75, 'DHR':140, 'MD':118},
    'KBG':{'JR':75, 'SCD':71},
    'SCD':{'KBG':71, 'DHR':151},
    'DHR':{'JR':140, 'SCD':151, 'SSD':80,
    'CMR':99},
    'CMR':{'DHR':99, 'GN':211},
    'GN':{'AGJ':101, 'CD':90, 'SC':85},
    'SC':{'GN':85, 'JS':98, 'DDM':142},
    'DDM':{'DAD':92, 'SC':142},
    'DAD':{'DDM':92, 'BHS':87},
    'BHS':{'DAD':87},
    'MD':{'JR':118,'SRM':111},
    'SRM':{'MD':111,'SSU':145},
    'SSU':{'SRM':145,'SD':120},
    'SD':{'SSU':120,'SSD':146,'AGJ':138},
    'SSD':{'DHR':80, 'SD':146, 'AGJ':97},
    'AGJ':{'SSD':97, 'SD':138, 'GN':101},
    'CD':{'GN':90},
    'JS':{'SC':98, 'GRD':86},
    'GRD':{'JS':86}}
```

Tree Search

- 문제의 solution은 action sequence를 search(검색) 함으로써 얻을 수있다. 이론적으로 모든 가능한 action sequence를 검색하면 항상 solution 혹은 optimal solution을 찾을 수 있다.
- Tree Search
  - 초기 상태부터 가능한 action을 선택해 가면서
  - 가능한 모든 action sequence를 tree로 만들어가는 기법
- Tree
  - 두 node를 잇는 path가 오직 1개씩만 존재하는 graph

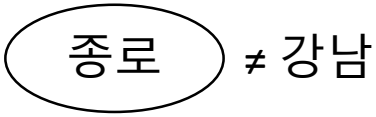


Tree Search

- Tree Search
  - 초기 상태부터 가능한 action을 선택해 가면서
  - 가능한 모든 action sequence를 tree로 만들어가는 기법

- Tree search의 의미
  - Node: agent가 가질 수 있는 하나의 state
  - Edge: state에서 취할 수 있는 액션 (또 다른 state에 연결)
  - Tree의 의미
    - 상위 node에서 하위 node로 이어지는 edge의 집합
    - 문제 해결을 위한 action sequence

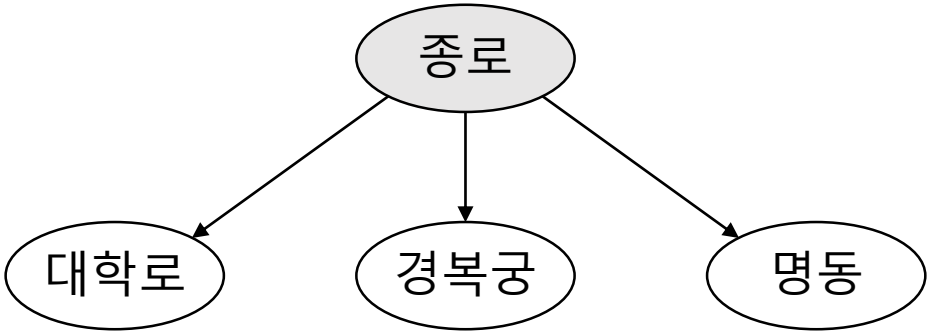
경로찾기 문제의 tree 표현



1) Goal test

- 현재 node의 state가 goal state인지 검사한다.
- Goal state가 맞다면 search를 종료하고, 아니라면 expansion을 시행한다.
- Initial state인 In(종로)는 당연히 goal state인 In(강남)이 아니므로 다른 state를 찾아가야 한다.

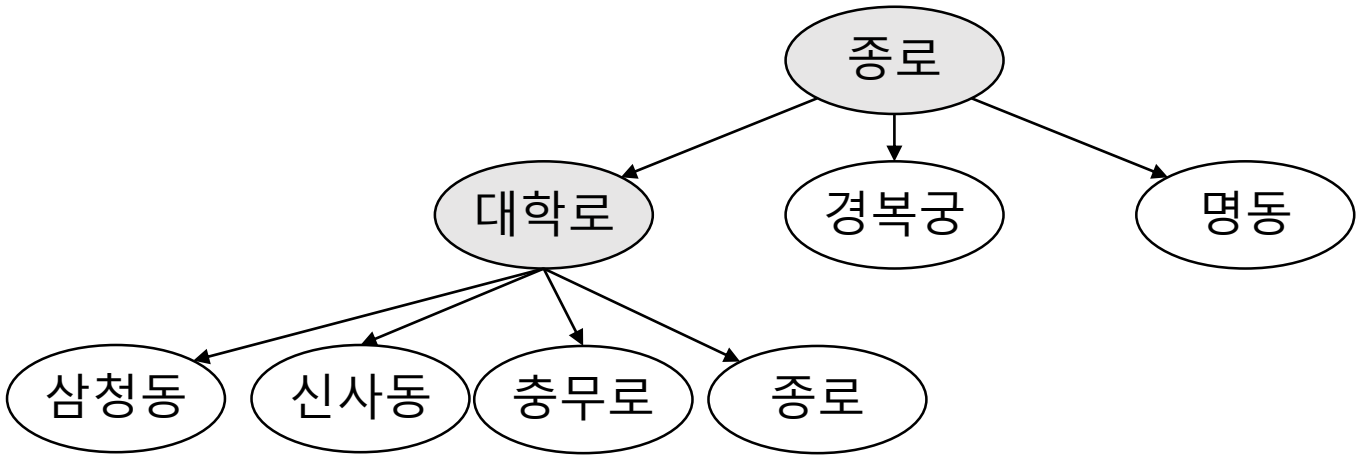
경로찾기 문제의 tree 표현



2) Expansion

- 현재 node에 연결되는 하위 node들을 분기해낸다.
- 현재 node의 state에서 선택할 수 있는 action의 결과로 만들어지는 state가 하위 node의 state가된다.
- 'In(종로)'에서 선택 가능한 action은 {Go(대학로), Go(경복궁),Go(명동)})'를 각각 state로 가지는 하위 node들이 생성된다.

경로찾기 문제의 tree 표현



2) Expansion

- 다음 state를 선택하고 (choosing), 해당 state가 goal을 달성했는지를 검사하고(testing), 달성치 못했다면 다시 expansion하는 과정을 반복하면서 tree가 커져간다(growing).
- 결국 goal state를 가진 node에 도달했을 때, root node부터 이 node까지의 edge의 집합, 즉 action sequence가 solution으로 얻어진다

Tree 표현을 위한 자료구조

- 각 노드(node)에서 다음 같은 정보를 가지도록 한다.

(1) State: node가 가지는 state

(2) Parent-node: 현재 node를 expansion한 상위 node

(3) Action: parent-node의 state에서 현재 node의 state로 이동하는 action

(4) Path-cost: 최초 root node에서 현재 node로 오기까지의 cost. Parent-node의 cost에 action의 step cost를 더함.

(5) Depth: 최초 root node에서 현재 node까지 단계(한 action이 한 단계) 수

- Node들의 집합으로 Tree를 표현할 수 있음.
  - Node: 자료형 안에 정의
  - Edge: parent-node 정보를 추적하여 얻을 수 있음

Search Tree 자료구조의 표현

그림 4의 예에서 보면 Node a와 b는 각각 'In(종로)'와 'In(대학로)'를 state로 가진다. Node b는 root node인 node a에서 분기된 node이다. 이 정보가 node b의 요소 중 parent node 값이 'a'라는 데이터로 표현된다. Node a와 b를 잇는 edge에 대응되는 action이 'Go(대학로)'라는 정보는 action 요소에 표현된다. Node a는 root node이므로 depth가 0이고, 여기에서 분기된 node b는 depth가 1이다. 종로에서 대학로까지의 거리가 140이므로 path cost는 140이다. 다시 node b에서 'Go(삼청동)' action을 선택하여 분기된 node c는 'In(삼청동)'을 state로 하고 b를 parent node로 한다. Depth는 1증가하여 2이며, path cost는 대학로에서 삼청동까지의 거리 151이 더해져 291이 된다.

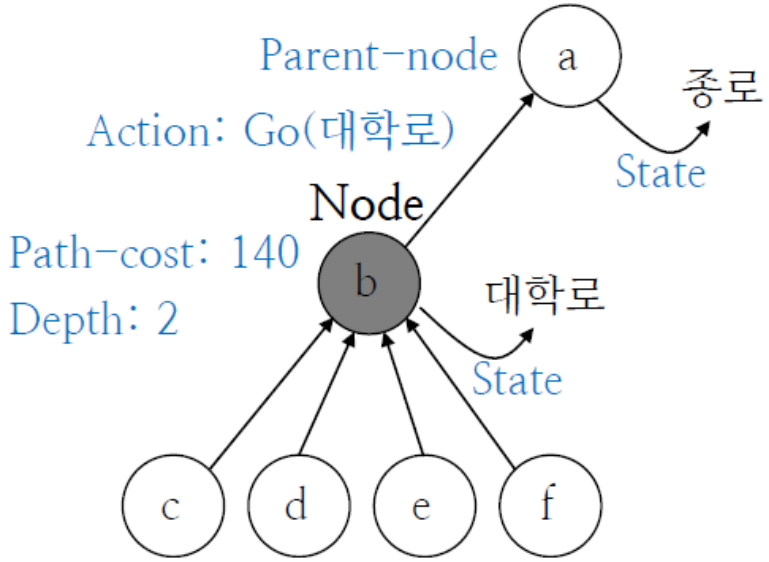
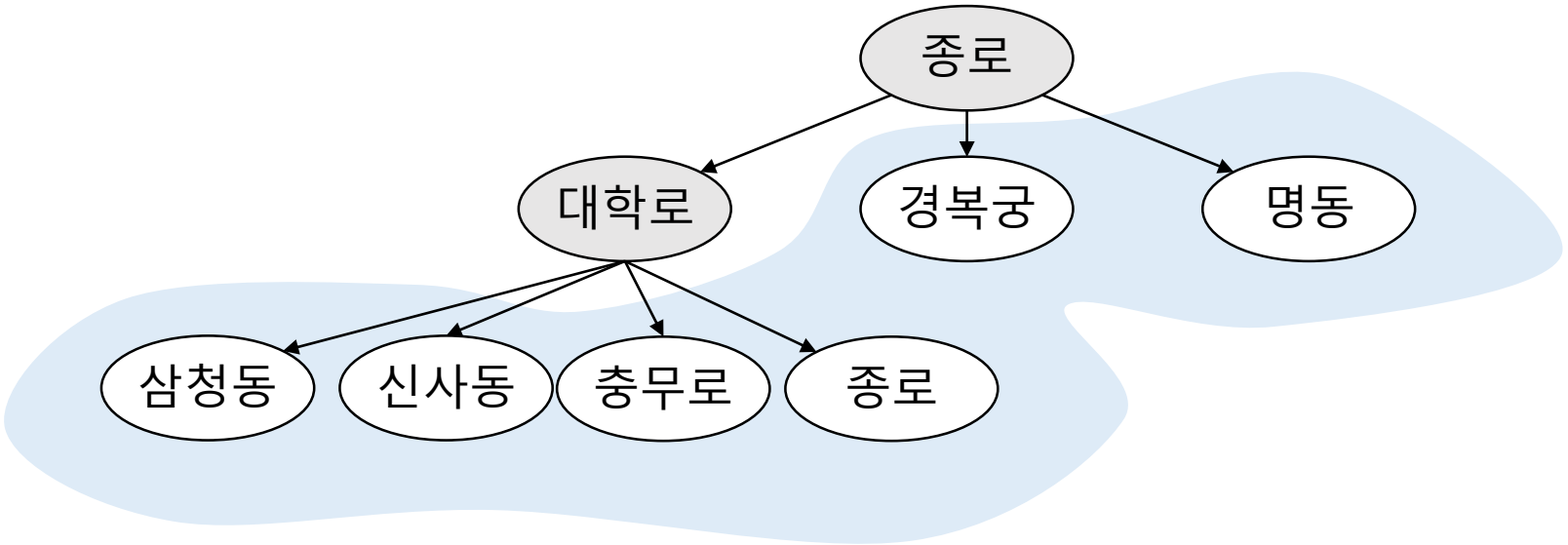


그림 4. Tree의 자료구조

Node	Depth	State	Parent node	Action	Path cost
a	0	In(종로)	–	–	0
b	1	In(대학로)	a	Go(대학로)	140
c	2	In(삼청동)	b	Go(삼청동)	291
d	2	In(충무로)	b	Go(충무로)	239
⋮	⋮	⋮	⋮	⋮	⋮

Fringe

- Tree에서 가장 바깥쪽에 위치하고 있는 노드들을 Fringe라고 함



- 우리는 매번 Fringe에 들어있는 노드 중 하나를 선택해나가야 한다.

실습_2: 에이전트 클래스 만들기

- 트리를 만들며 탐색을 수행하는 Agent를 클래스로 설계
  - 클래스 변수
    - goal_state: 입력된 목표도시
    - graph: 저장할 그래프
    - n_node: 트리에 추가된 마지막 노드 번호
    - fringe: 프린지 (노드 번호로만 표현)
    - visited: 선택된 노드 (노드 번호로만 표현)
    - finished: 목표를 찾았는지 여부
  - 클래스 메소드
    - get_child_node: 해당 노드에서 뻗어나갈 수 있는 하위 노드들 탐색
    - expansion(노드번호): 노드를 선택하여 확장수행
      - fringe에서 해당노드 제거
      - visited에 해당노드 추가
      - get_child_node에서 생성된 하위노드들 트리에 추가
      - fringe에 하위노드들 추가
    - get_path(노드번호): 해당 노드번호까지 방문한 도시들 호출
    - print_state(): 현재 에이전트의 상태변수들 출력

실습_2:에이전트 클래스 만들기

```python
class agent:
    def __init__(self, initial_state, goal_state):
        self.goal_state=goal_state #목표도시
        self.graph={0:{'depth':0,'state':initial_state,'parent':None,'cost'
        self.n_node=0 # 그래프에 추가된 마지막 노드 번호
        self.fringe=[0] # 프린지 (첫번째 노드는 무조건 추가)
        self.visited=[] # 방문한 노드
        self.finished=False # 목표를 찾았는지 여부

    # 해당 노드에서 뻗어나갈 수 있는 하위 노드들 생성
    def get_child_node(self,gid):
        result=[]
        city=self.graph[gid]['state'] #선택한 노드 아이디의 도시 선택
        child_cities=route_actions[city].keys() #선택한 노드에서 뻗칠수 있는
        for child in child_cities:
            ## 이부분을 완성해 보시오

        return result

    # 노드를 선택하여 확장
    # 프린지에서 해당노드는제거
    # 선택한 노드에 추가
    # 그래프에 아이디 하나씩 추가하여 생성
    def expansion(self,gid):
        if gid in self.fringe:
            self.fringe.remove(gid) #프린지에서 해당노드제거
            self.visited.append(gid) #방문노드에 해당노드추가
            ## 이부분을 완성해 보시오

        else:
            print('only nodes in fringe can be expanded')

    # 처음부터 해당 노드까지 경로 (도시방문경로)
    def get_path(self,gid):
        result=[]
        parent=self.graph[gid]['parent']
        while parent!=None:
            result.append(self.graph[parent]['state'])
            parent=self.graph[parent]['parent']
        return result

    def print_state(self):
        print('graph:',self.graph)
        print('fringe:',self.fringe)
        print('visited:',self.visited)
```

실습_2: 에이전트 클래스 만들기

```
my_agent=agent('JR','GN')
my_agent.print_state()
```

```
graph: {0: {'depth': 0, 'state': 'JR', 'parent': None, 'cost': 0}}
fringe: [0]
visited: []
n_node: 0
finished: False
```

```
my_agent.get_child_node(0)
```

```
[{'depth': 1, 'state': 'KBG', 'parent': 0, 'cost': 75},
 {'depth': 1, 'state': 'DHR', 'parent': 0, 'cost': 140},
 {'depth': 1, 'state': 'MD', 'parent': 0, 'cost': 118}]
```

```
my_agent.expansion(0)
my_agent.print_state()
```

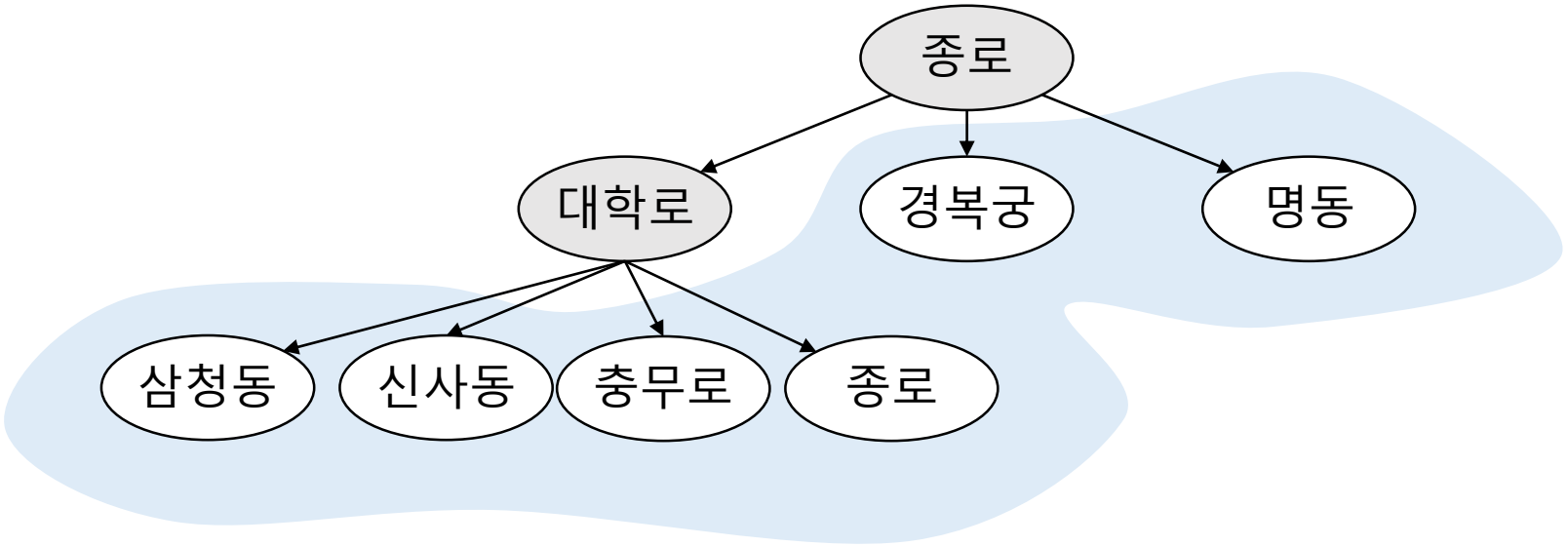
```
graph: {0: {'depth': 0, 'state': 'JR', 'parent': None, 'cost': 0}, 1: {'depth':
1, 'state': 'KBG', 'parent': 0, 'cost': 75}, 2: {'depth': 1, 'state': 'DHR', 'pa
rent': 0, 'cost': 140}, 3: {'depth': 1, 'state': 'MD', 'parent': 0, 'cost': 11
8}}
fringe: [1, 2, 3]
visited: [0]
n_node: 3
finished: False
```

```
my_agent.expansion(2)
my_agent.print_state()
```

```
graph: {0: {'depth': 0, 'state': 'JR', 'parent': None, 'cost': 0}, 1: {'depth':
1, 'state': 'KBG', 'parent': 0, 'cost': 75}, 2: {'depth': 1, 'state': 'DHR', 'pa
rent': 0, 'cost': 140}, 3: {'depth': 1, 'state': 'MD', 'parent': 0, 'cost': 11
8}, 4: {'depth': 2, 'state': 'JR', 'parent': 2, 'cost': 280}, 5: {'depth': 2, 's
tate': 'SCD', 'parent': 2, 'cost': 291}, 6: {'depth': 2, 'state': 'SSD', 'paren
t': 2, 'cost': 220}, 7: {'depth': 2, 'state': 'CMR', 'parent': 2, 'cost': 239}}
fringe: [1, 3, 4, 5, 6, 7]
visited: [0, 2]
n_node: 7
finished: False
```

Search Strategy

- Search Strategy: Tree에서 어떤 노드를 골라서 expansion해 나갈 것인가?
- Fringe 중 어느 node에서 다음 목적지를 찾을 것인지?



알고리즘 (algorithm)

- Search strategy에 따라 solution을 탐색하는 과정
- 알고리즘의 성능을 판단하는 기준
- Optimality와 complexity는 trade-off관계에 있음

(1) Completeness
Algorithm에 따라 search를 수행하면 solution을 찾을 수 있는가? (solution이 존재하는 경우에 한해)

(2) Optimality
Algorithm이 찾은 solution이 optimal solution (모든 solution 중 path-cost가 가장 작은 solution)인가?

(3) Time complexity
Solution을 찾을 때까지 얼마나 많은 시간이 걸리는가? (시간은 계산 횟수와 같음)

(4) Space complexity
Search를 수행하기 위해 얼마나 많은 저장공간(컴퓨터 memory)이 필요한가?

Uninformed Search Strategy

- Search Strategy
  - Fringe에서 어떤 방향으로 node를 펼처나 갈 것인가?

- Uninformed Search Strategy
  - '현재 위치' 라는 단순한 정보만을 이용하여 탐색하는 방법

(1) Breadth-first Search
Node를 횡으로(breadth) 먼저 탐색하는 전략. 가장 depth가 얕은 node부터 expansion한다.

(2) Depth-first Search
Node를 종으로(depth) 먼저 탐색하는 전략. 가장 depth가 깊은 node부터 expansion한다.

(3) Iterative Deepening Search
Depth-first search에서 종으로 내려갈 depth의 한계(limit)를 정해 그 이상 내려가지 않되, solution이 찾아지지 않으면 이 limit를 계속 키워가며 탐색하는 전략.

Breadth-first search (너비 우선 탐색)

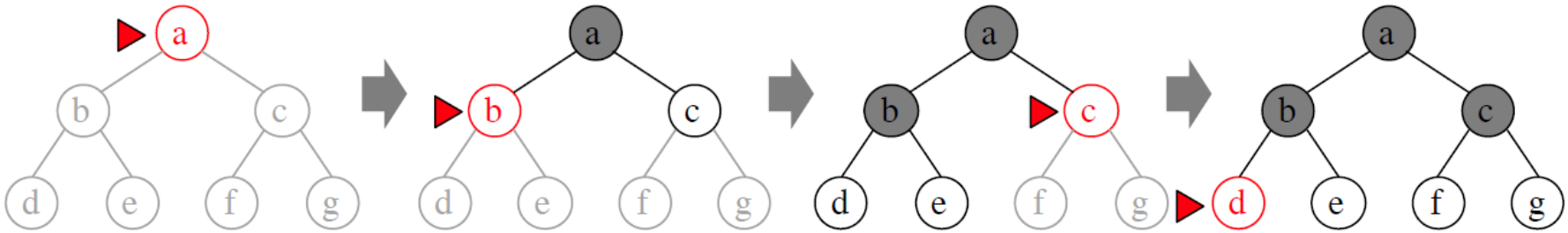
- Depth가 가장 낮은 state들을 모두 expansion 한 후, 다음 depth로 넘어간다.

(1) Initial state에 해당하는 root node를 expansion하여 하위 node들을 생성한다.

(2) Expansion 가능한 node들(fringe) 중 depth가 가장 얕은 node를 expansion한다.

(3) 위 (2)의 과정을 solution을 찾을 때까지, 혹은 더 이상 expansion할 node가 남아있지 않을 때까지 계속한다.

그림 5의 예시를 보면 (1)에 의해 root node a를 expansion하고, (2)에 의해 depth가 1일 node b를 expansion하여 d와 e를 생성한다. 다음 다시 (2)에 의해 c, d, e 중 depth가 가장 낮은 c를 expansion하여 f와 g를 생성한다. 이를 다시 fringe인 d, e, f, g에 대해 반복한다.



Breadth-first Search 알고리즘 성능

- Completeness
  - 완전한 해를 찾을 수 있는가? → Yes

- Optimality
  - 최적의 해를 찾을 수 있는가? → Not always
  - 더 깊은 곳에, 더 좋은 해 존재 가능
    - 대학로>충무로>강남
    - 대학로>신사>압구정>강남

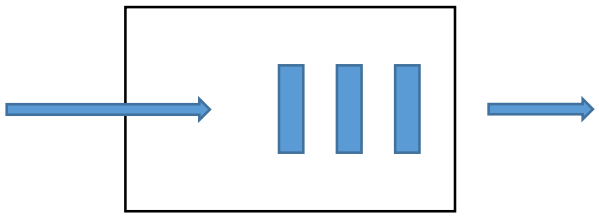
Breadth-first Search 알고리즘 성능

- Space complexity
  - 한 노드에서 expansion되는 하위 노드의 수를 b개라고 할 때, n번째 depth까지 확장하면 필요한 저장 공간은 다음과 같음
    $b + b^2 + \ldots + b^n + b^{n+1} = o(b^{n+1})$
  - 한 노드에 10byte가 필요하고 depth가 10이면, 1terabyte 필요!

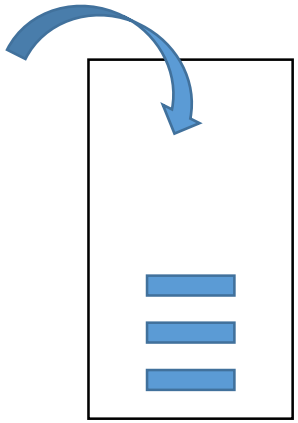
- Time complexity
  - 생성되는 모든 node에 대해 goal test와 expansion이 필요하므로, space complexity와 같은 복잡도 발생

Reminde: 자료구조 (큐 vs 스택)

큐(queue):
First-in-first-out



스택(stack): last-in-first-out



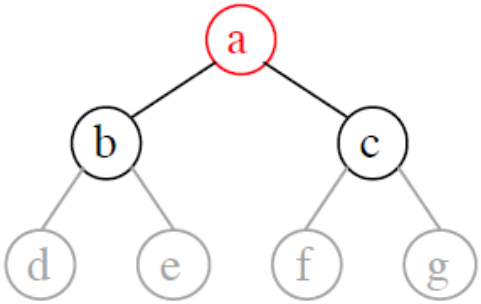
BFS 알고리즘 구현: 큐를 이용

0) Queue에 a를 넣고

→ a

1) a를 꺼냄

→ a

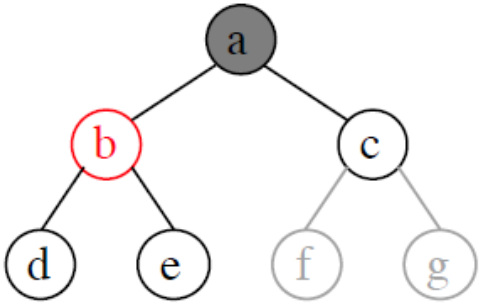


2) b,c를 넣음

→ c b

3) b를 꺼냄

c → b

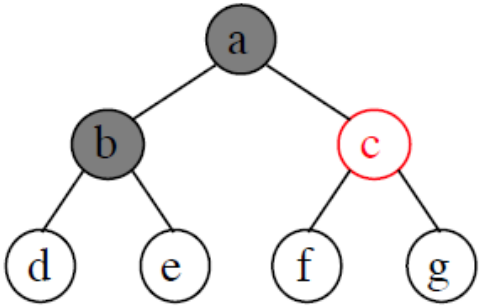


4) d,e를 넣음

→ e d c

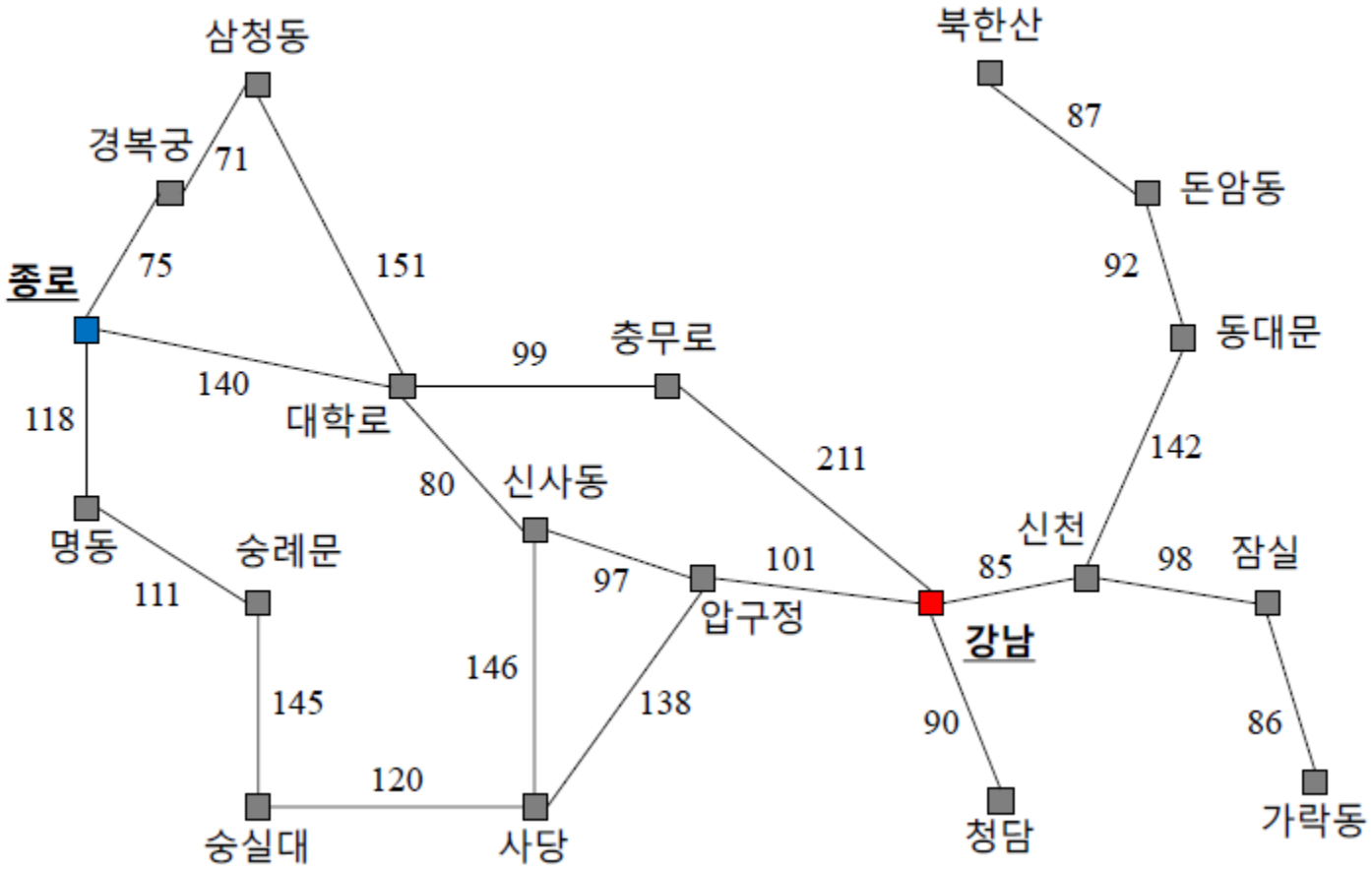
5) c를 꺼냄

e d → c



실습_1: BFS

다음 지도를 참고해서, 종로부터 강남으로 가는 BFS 알고리즘의 TREE를 손으로 그려보시오.



BFS 알고리즘

```
def bf_search_basic(actions, init_state, goal_state):
    root = {'depth': 0, 'state': init_state, 'parent': -1, 'action': (init_state, 0), 'cost': 0}
    fringe = [root] # 프린지
    tree = {} # 만들어낼 tree
    next_id = 0 #노드의 id

    while len(fringe) > 0:
        this_node = fringe.pop() #큐로부터 하나씩 빼기

        # 큐로부터 빠진 노드(this_node)에 id 부과하기
        this_id = next_id
        # 트리에 아이디와 함께 해당 노드 추가
        tree[this_id] = this_node
        # 다음 루프를 위해 next_id에 1 추가
        next_id += 1

        # 만약 이 노드가 목표하는 노드였다면?
        if this_node['state'] == goal_state:
            return (this_id, tree) #종료하면서 트리와 아이디 리턴

        # 아니라면, 이 노드를 expansion
        children = expansion(this_id, this_node, actions)
        fringe = children + fringe

    return (-1, tree)
```

BFS 알고리즘 구현: 중복 방지

- 방문했던 도시를 또 방문하는 현상 발생 가능
  - 같은 Tree가 똑같이 반복되는 현상 발생

- 해결책
  - 이미 탐색된 노드들의 정보를 저장
  - Path: Root 노드부터 해당 노드까지 선택된 노드들
  - 새로 탐색할 node가 path에 이미 존재한다면, 탐색을 중단

중복방지 function 추가

```
def bf_search(actions, init_state, goal_state):
    root = {'depth': 0, 'state': init_state, 'parent': -1, 'action': (init_state, 0), 'cost': 0}
    fringe = [root] # 프린지
    tree = {} # 만들어낼 tree
    next_id = 0 #노드의 id

    while len(fringe) > 0:
        this_node = fringe.pop() #큐로부터 하나씩 빼기

        # 큐로부터 빠진 노드(this_node)에 id 부과하기
        this_id = next_id
        # 트리에 아이디와 함께 해당 노드 추가
        tree[this_id] = this_node
        # 다음 루프를 위해 next_id에 1 추가
        next_id += 1

        # 만약 이 노드가 지금까지 거쳐왔던 path안에 속해있었다면 중복이므로 확장하지 않는다.
        if this_node['state'] in trace_path(this_node['parent'], tree):
            continue
        else: # 아니라면
            if this_node['state'] == goal_state: #만약 이 노드가 목표하는 노드였다면?
                return (this_id, tree) #종료하면서 트리와 아이디 리턴
            # 아니라면, 이 노드를 expansion
            children = expansion(this_id, this_node, actions)
            fringe = children + fringe # 추가되는 순서를 잘 살펴보라

    return (-1, tree)

def trace_path(node_id, tree): #이 노드로부터 루트노드까지의 path 리턴
    path = []
    while node_id>=0:
        node = tree[node_id]
        path.append(node['state'])
        node_id = node['parent']

    return path
```

Depth-first Search

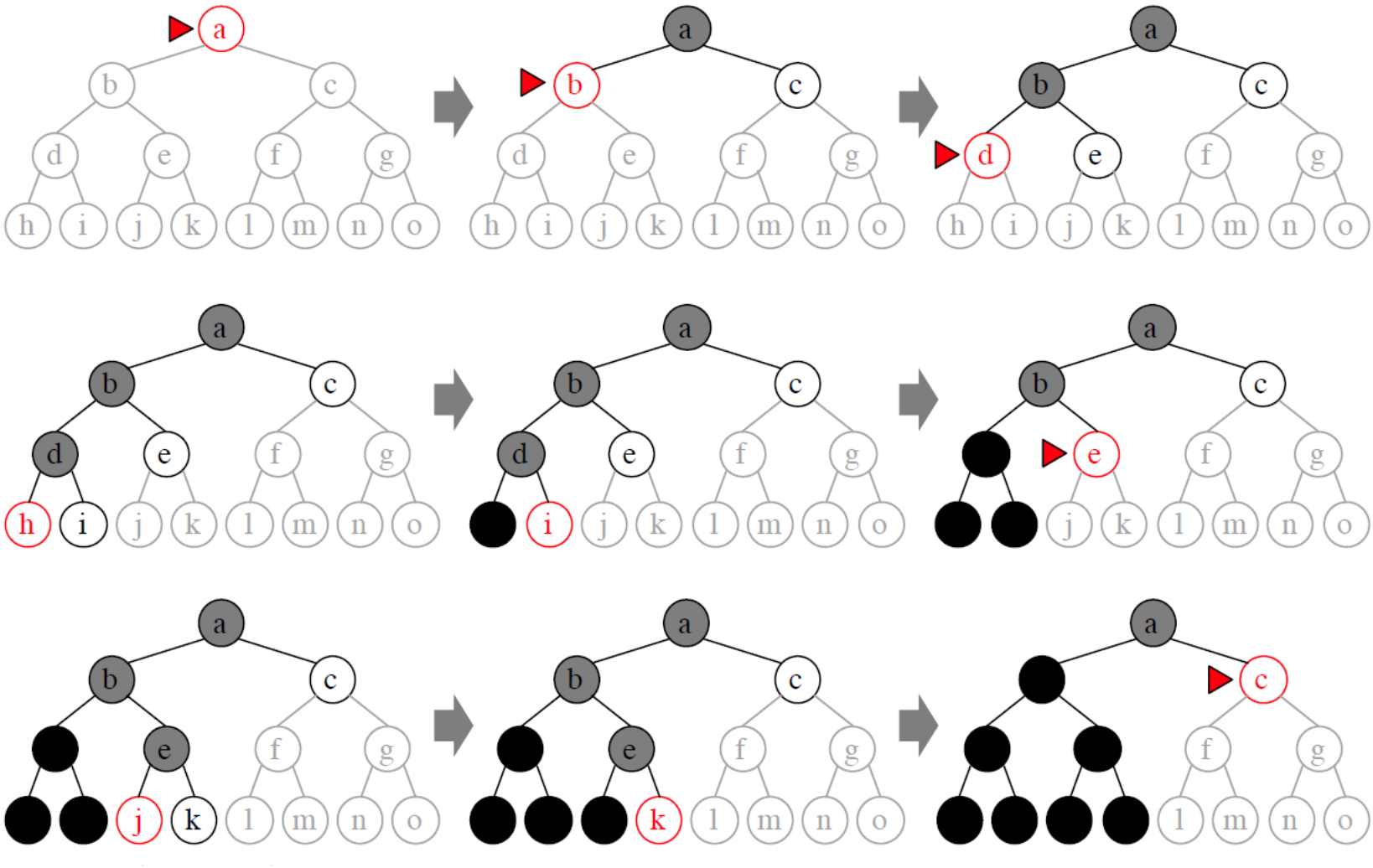
- 현재까지 확장된 node 중 depth가 가장 깊은 node를 먼저 확장

(1) Initial state에 해당하는 root node를 expansion하여 하위 node들을 생성한다.

(2) Expansion 가능한 node들(fringe) 중 depth가 가장 깊은 node를 expanding한다.

(3) 위 (2)의 과정을 solution을 찾을 때까지, 혹은 더 이상 expansion할 node가 남아있지 않을 때까지 계속한다.

Depth-first Search



DFS 알고리즘의 성능

- Completeness
  - 완전한 해를 찾을 수 있는가? → Not Complete
  - 무한히 내려갈 수 있다

- Optimality
  - 최적의 해를 찾을 수 있는가? → Not always
  - 탐색하지 않은곳에 최적해가 있을 수 있다.

DFS 알고리즘의 성능

- Space complexity
  - 한 노드에서 expansion되는 하위 노드의 수를 b개라고 할 때, n번째 depth까지 확장하면 필요한 저장 공간은 다음과 같음
  - 한 노드에 10byte가 필요하고 depth가 10이면, 1terabyte 필요!
- Time complexity
  - 생성되는 모든 node에 대해 goal test와 expansion이 필요하므로, space complexity와 같은 복잡도 발생

스택을 이용한 DFS 알고리즘

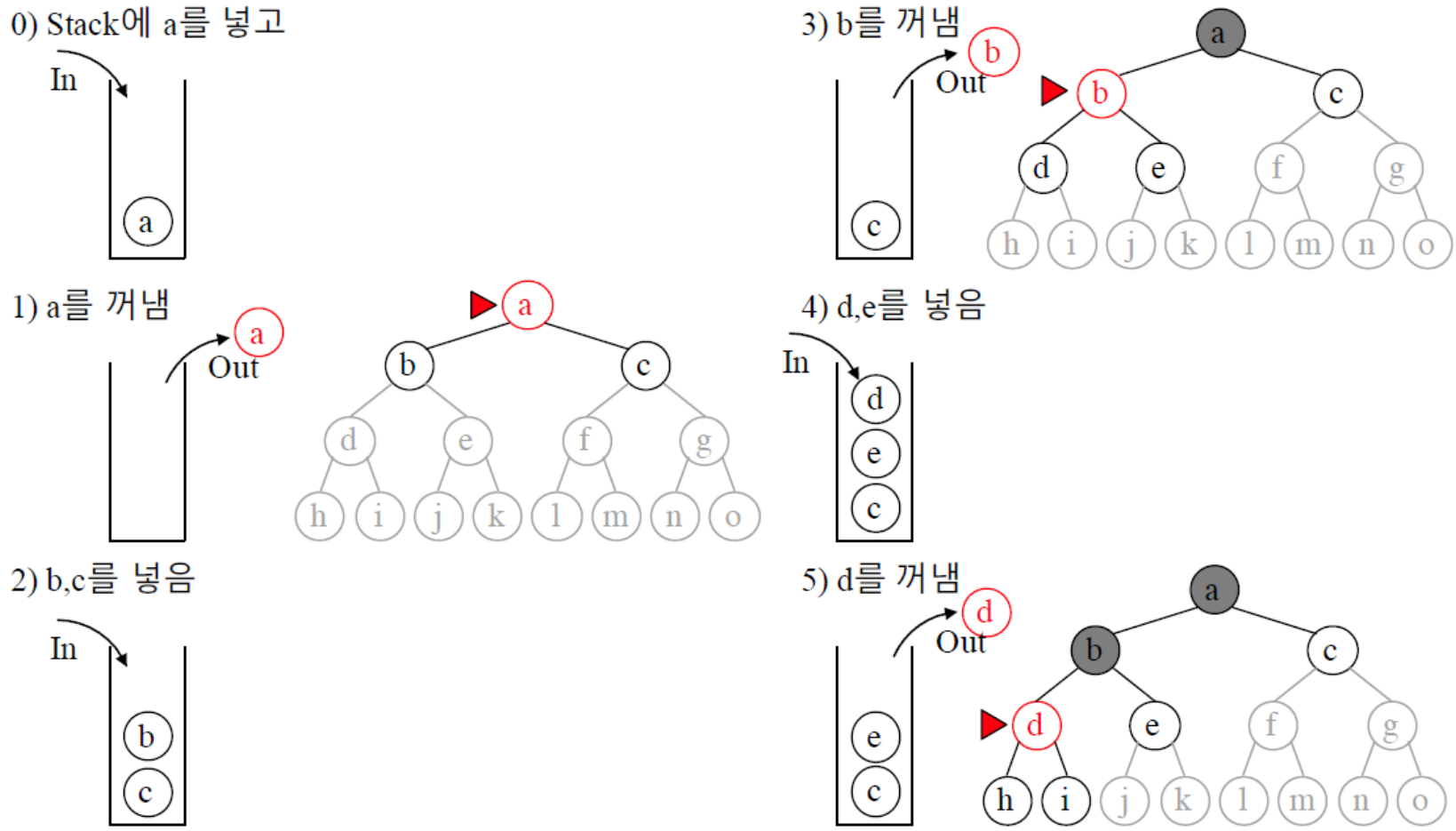


그림 8. Stack을 이용한 depth-first search

과제:

- BFS알고리즘을 변형하여
- DFS알고리즘을 구현!