인공지능개론 과제

Q1. 균일 비용 탐색 알고리즘이 뭔지 설명해줘.

A1. 균일 비용 탐색 알고리즘(Uniform Cost Search, UCS)은 그래프의 정점 사이의

가중치가 모두 동일한 경우에 사용되는 그래프 탐색 알고리즘입니다. 이 알고리즘은

다익스트라 알고리즘의 특별한 경우로 볼 수 있습니다.

균일 비용 탐색 알고리즘은 너비 우선 탐색(BFS)을 확장한 것으로, 각 단계에서 가장

적은 비용을 가진 경로를 확장하는 방식으로 동작합니다. 이 알고리즘은 목표 지점에

도달할 때까지 계속 경로를 확장하면서 최소 비용의 경로를 찾습니다. 따라서 경로의

비용이 낮은 경우에 더 빨리 탐색이 가능합니다.

균일 비용 탐색 알고리즘은 일반적으로 우선순위 큐를 사용하여 구현됩니다. 각 정점은 해당 정점까지의 경로 비용을 기준으로 우선순위 큐에 저장되며, 가장 비용이 적은

정점부터 선택하여 탐색을 진행합니다.

이 알고리즘은 최적의 해를 보장하지만, 그래프의 크기가 커질수록 탐색 시간이 증가할 수 있습니다. 또한, 모든 간선의 가중치가 동일하지 않은 경우에는 다른 알고리즘을

사용해야 합니다.

Q2. 이 코드를 설명해줘.

**function UNIFORM-COST-SEARCH(problem) returns a solution, or failure**

**node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0**

**frontier ← a priority queue ordered by PATH-COST, with node as the only element**

**explored ← an empty set**

**loop do**

**if EMPTY?(frontier) then return failure**

**node ← POP(frontier) /\* chooses the lowest-cost node in frontier \*/**

**If problem.GOAL-TEST(node.STATE) then return SOLUTION(node)**

**add node.STATE to explored**

**for each action in problem.ACTIONS(node.STATE) do**

**child ← CHILD-NODE(problem,node,action)**

**if child.STATE is not in explored or frontier then**

**frontier ← INSERT(child,frontier)**

**else if child.STATE is in frontier with higher PATH-COST then**

**replace that frontier node with child**

A2. 이 코드는 균일 비용 탐색(Uniform Cost Search) 알고리즘의 구현입니다. 이 알고리즘은 경로 비용이 각각 다른 경우에도 사용할 수 있는 일반적인 그래프 탐색 알고리즘입니다. 코드의 각 부분을 설명하겠습니다.

1. function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

균일 비용 탐색 알고리즘의 메인 함수로, 문제를 입력으로 받고 해결책이나 실패를 반환합니다.

2. node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0

초기 노드를 생성합니다. 이 노드는 문제의 초기 상태를 가지며, 경로 비용은 0입니다.

3. frontier ← a priority queue ordered by PATH-COST, with node as the only element

경로 비용을 기준으로 정렬된 우선순위 큐인 frontier를 초기화합니다. 초기 노드를 frontier에

추가합니다.

4. explored ← an empty set

이미 탐색한 노드들의 상태를 저장할 빈 집합을 생성합니다.

5. loop do

무한 루프를 시작합니다.

6. if EMPTY?(frontier) then return failure

frontier가 비어있다면 실패를 반환합니다. 즉, 더 이상 탐색할 노드가 없는 경우입니다.

7. node ← POP(frontier) /\* chooses the lowest-cost node in frontier \*/: frontier에서 가장 경로

비용이 작은 노드를 가져옵니다.

8. If problem.GOAL-TEST(node.STATE) then return SOLUTION(node)

가져온 노드의 상태가 목표 상태인지 검사합니다. 목표 상태라면 해당 노드를 해결책으로

반환합니다.

9. add node.STATE to explored

현재 노드의 상태를 이미 탐색한 노드들의 집합에 추가합니다.

10. for each action in problem.ACTIONS(node.STATE) do

현재 상태에서 가능한 각 행동에 대해 반복합니다.

11. child ← CHILD-NODE(problem, node, action)

현재 상태에서 주어진 행동을 수행한 후의 자식 노드를 생성합니다.

12. if child.STATE is not in explored or frontier then

자식 노드의 상태가 이미 탐색한 노드들의 집합에도 없고 frontier에도 없는 경우:

frontier ← INSERT(child, frontier): 자식 노드를 frontier에 추가합니다.

13. else if child.STATE is in frontier with higher PATH-COST then

이미 frontier에 있는 경우이고, 그 경로 비용이 더 큰 경우:

replace that frontier node with child: 해당 frontier 노드를 자식 노드로 대체합니다. 즉, 경로 비용이 더 작은 새로운 경로를 반영합니다.

이렇게 하여 알고리즘이 계속해서 경로를 확장하고, 목표 상태에 도달하거나 더 이상 확장 가능한 경로가 없을 때까지 반복하여 해결책을 찾습니다.

Q3. 위 코드를 파이썬으로 구현해줘.

A3. 아래는 주어진 균일 비용 탐색 알고리즘을 파이썬으로 구현한 코드입니다.

**import heapq**

**def uniform\_cost\_search(problem):**

**node = (problem.initial\_state, 0) # (STATE, PATH-COST)**

**frontier = [(0, node)] # (PATH-COST, (STATE, PATH-COST))**

**explored = set()**

**while frontier:**

**\_, (state, path\_cost) = heapq.heappop(frontier)**

**if problem.goal\_test(state):**

**return state # 해결책 반환**

**explored.add(state)**

**for action in problem.actions(state):**

**child = problem.child\_node(state, action)**

**child\_state, child\_path\_cost = child**

**if child\_state not in explored and not any(child\_state == node[1][0] for \_, node in frontier):**

**heapq.heappush(frontier, (child\_path\_cost, child))**

**elif any(child\_state == node[1][0] and child\_path\_cost < path\_cost for \_, node in frontier):**

**frontier = [(p, n) if n[1][0] != child\_state else (child\_path\_cost, child) for p, n in frontier]**

**return None # 실패 시 None 반환**

위 코드에서는 다음과 같은 전제를 가정합니다

* **problem** 객체는 균일 비용 탐색에 필요한 메서드를 제공합니다.
* **problem.initial\_state**는 초기 상태를 반환합니다.
* **problem.goal\_test(state)**는 주어진 상태가 목표 상태인지 여부를 반환합니다.
* **problem.actions(state)**는 주어진 상태에서 가능한 모든 행동을 반환합니다.
* **problem.child\_node(state, action)**는 주어진 상태와 행동에 대한 자식 노드를 반환합니다. 이 함수는 (자식 상태, 자식 경로 비용) 형식의 튜플을 반환해야 합니다.

이 코드를 사용하려면 적절한 **problem** 객체를 정의해야 합니다. 이 객체는 위에 설명된 메서드를 구현하여야 합니다.

Q4. 파이썬에 사용된 코드는 무슨 의미인지 설명해줘

**A4.** 파이썬으로 구현된 코드는 주어진 균일 비용 탐색 알고리즘을 구현한 것입니다. 여기서 각 부분에 대한 설명을 제공하겠습니다

**1. import heapq**

파이썬의 힙 큐(heap queue) 모듈을 가져옵니다. 이 모듈은 최소 힙(min heap)을 지원하며, 우선순위 큐를 구현하는 데 사용됩니다.

**2. def uniform\_cost\_search(problem)**

균일 비용 탐색 알고리즘을 수행하는 함수를 정의합니다. 이 함수는 **problem** 객체를 입력으로 받습니다.

**3. node = (problem.initial\_state, 0)**

초기 노드를 생성합니다. 이 노드는 문제의 초기 상태와 해당 상태까지의 경로 비용(0)을 포함한 튜플입니다

**4. frontier = [(0, node)]**

우선순위 큐인 **frontier**를 초기화합니다. 이 큐는 경로 비용을 기준으로 정렬되어 있으며, 초기 노드를 포함합니다. 각 원소는 경로 비용과 노드를 포함하는 튜플입니다.

**5. explored = set()**

이미 탐색한 노드들의 상태를 저장할 집합을 초기화합니다.

**6. while frontier**

**frontier**가 빌 때까지 무한 루프를 시작합니다.

**7. \_, (state, path\_cost) = heapq.heappop(frontier)**

우선순위 큐인 **frontier**에서 경로 비용이 가장 작은 노드를 가져옵니다. 이 노드의 상태와

경로 비용을 추출합니다.

**8. if problem.goal\_test(state)**

가져온 노드의 상태가 목표 상태인지 검사합니다. 만약 그렇다면 해당 상태를 반환하고

알고리즘을 종료합니다.

9. **explored.add(state)**

현재 노드의 상태를 이미 탐색한 상태들의 집합에 추가합니다.

10. **for action in problem.actions(state)**

현재 상태에서 가능한 각 행동에 대해 반복합니다.

11. **child = problem.child\_node(state, action)**

주어진 상태와 행동에 대한 자식 노드를 생성합니다. 이 함수는 자식 상태와 그 상태까지의 경로 비용을 반환합니다.

12. 자식 노드의 상태가 탐색되지 않았거나 **frontier**에 없는 경우, 자식 노드를 **frontier**에

추가합니다.

**heapq.heappush(frontier, (child\_path\_cost, child))**: 우선순위 큐인 **frontier**에 자식 노드를

추가합니다.

13. 이미 탐색된 노드이거나 **frontier**에 있는 경우, 더 낮은 경로 비용을 가진 노드로

업데이트합니다.

**frontier = [(p, n) if n[1][0] != child\_state else (child\_path\_cost, child) for p, n in frontier]**: **frontier**에 있는 해당 노드의 경로 비용을 업데이트합니다.

14. 알고리즘이 종료되면, 실패를 의미하는 **None**을 반환합니다.

이 코드는 파이썬의 리스트와 우선순위 큐를 활용하여 균일 비용 탐색 알고리즘을 구현합니다.