컴퓨터인공지능학부 202312720 안수빈

1. search.py의 depth_first_search, breadth_first_search, uniform_cost_search, aStar_search 및 heuristic 함수 알고리즘 동작 방식에 대한 설명

1) DFS: Depth-First Search (깊이 우선 탐색)

DFS는 스택을 사용하여 탐색을 수행합니다. 현재 노드에서 갈 수 있는 곳까지 최대한 깊이 탐색한 후, 더 이상 갈 속이 없으면 다시 뒤로 돌아오는 백트래킹을 수행합니다. search.py에서 DFS는 visited 집합을 사용하여 이미 한번 방문한 노드는 다시 재방문하지 않도록 하고 있습니다. DFS는 메모리 최대 탐색 깊이에 비례하여 메모리 사용량이 적다는 장점이 있습니다. 하지만 최적해를 보장하지 않고, 만약 깊은 탐색이 필요한 경우 속도가 느려질 수 있다는 단점이 있습니다.

2) BFS: Breadth-First Search (너비 우선 탐색)

BFS는 큐를 사용하여 구현합니다. 현재 깊이의 모든 노드를 탐색한 후에 다음 깊이의 노드를 탐색하는 방식으로 수행됩니다. search.py에서는 방문한 노드를 visited 집합에 저장하여 다시 재방문하는 것을 방지합니다. 또 BFS는 모든 비용이 같은 경우 최단 경로를 보장합니다. BFS는 모든 비용이 같은 경우 최단 경로를 보장하고 가까운 곳부터 탐색하므로 문제에 대한 정답을 찾을 확률이 높다는 장점이 있습니다. 하지만 탐색할 노드를 큐에 저장해야 하므로 메모리 사용량이 많다는 단점이 있습니다.

3) UCS: Uniform Cost Search (균일 비용 탐색)

UCS는 우선순위 큐를 사용하여 현재의 비용(g(n))이 가장 작은 노드를 먼저 탐색합니다. BFS와 비슷하지만, 비용(cost)이 적은 경로를 먼저 탐색하는 점이 다릅니다. UCS도 다른 알고리즘과 마찬가지로 visited 집합을 사용하여 노드의 재방문을 방지합니다. UCS는 최적 경로를 보장하고 비용(cost)이 다양한 경우 BFS보다 효율적으로 사용할 수 있다는 장점이 있습니다. 하지만 우선순위 큐 정렬로 인해 속도가 느려질 수 있고 메모리 사용량이 많다는 단점이 있습니다.

4) A* 알고리즘

A*는 휴리스틱 함수를 추가하여 UCS보다 효율적으로 탐색합니다.

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

g(n): 출발점에서 현재 상태까지의 비용 (실제 이동한 거리)

h(n): 현재 상태에서 목표 상태까지의 추정 비용 (휴리스틱)

휴리스틱 함수가 admissible(과대평가하지 않음)하고 consistent(단조적)해야 A*가 최적해를 보장할 수 있습니다. astar는 적절한 휴리스틱을 사용할 때 최적 경로를 보장하고 UC보다 더 빠른 속도로 탐색이 가능하다는 장점이 있습니다. 하지만 휴리스틱이 적절하지 않을 때는 비효율적이고 메모리 사용량이 많다는 단점이 있습니다.

2. 아래는 각 알고리즘을 시간과 경로로 비교 분석해보기 위해서 moves = 100으로 10회 실행시킨 결과를 요약해 놓은 것입니다.

[1차 실행]

<rand> 이동 횟수/소요 시간: 14 moves / Execution time: 1.9173 seconds

<DFS> 이동 횟수/소요 시간: 47978 moves / Execution time: 72.1357 seconds

<BFS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0953 seconds
<A*> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0050 seconds
<UCS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0955 seconds

breadth_first_search found a path of 14 moves: ['down', 'right', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'left', 'down', 'right']

aStar_search found a path of 14 moves: ['down', 'right', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'left', 'down', 'right']

uniform_cost_search found a path of 14 moves: ['down', 'right', 'down', 'right', 'up', 'up', 'left', 'down', 'down', 'right', 'up', 'left', 'down', 'right']

[2차 실행]
<rand> 이동 횟수/소요 시간: 20 moves / Execution time: 1.0000 seconds
<DFS> 이동 횟수/소요 시간: 69560 moves / Execution time: 194.7767 seconds
<BFS> 이동 횟수/소요 시간: 16 moves / Execution time: 0.3530 seconds
<A*> 이동 횟수/소요 시간: 16 moves / Execution time: 0.0348 seconds
<UCS> 이동 횟수/소요 시간: 16 moves / Execution time: 0.3888 seconds

breadth_first_search found a path of 16 moves: ['down', 'left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'right']

aStar_search found a path of 16 moves: ['down', 'left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'right']

uniform_cost_search found a path of 16 moves: ['down', 'left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'up', 'left', 'down', 'down', 'right', 'right']

[3차 실행]
<rand> 이동 횟수/소요 시간: 14 moves / Execution time: 0.7794 seconds
<DFS> 이동 횟수/소요 시간: 21026 moves / Execution time: 12.4198 seconds
<BFS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.1016 seconds
<A*> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0063 seconds
<UCS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.1134 seconds

breadth_first_search found a path of 14 moves: ['down', 'right', 'down', 'left', 'up', 'up', 'right', 'down', 'down', 'right', 'up', 'left', 'down', 'right']

aStar_search found a path of 14 moves: ['down', 'right', 'down', 'left', 'up', 'up', 'right', 'down', 'down', 'right', 'up', 'left', 'down', 'right']

uniform_cost_search found a path of 14 moves: ['down', 'right', 'down', 'left', 'up', 'up', 'right', 'down', 'down', 'right', 'up', 'left', 'down', 'right']

[4차 실행]
<rand> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0174 seconds
<DFS> 이동 횟수/소요 시간: 24792 moves / Execution time: 18.9792 seconds
<BFS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0269 seconds
<A*> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0013 seconds
<UCS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0238 seconds

breadth_first_search found a path of 10 moves: ['right', 'up', 'left', 'up', 'right', 'down', 'left', 'down', 'right', 'right']

aStar_search found a path of 10 moves: ['right', 'up', 'left', 'up', 'right', 'down', 'left', 'down', 'right', 'right']

uniform_cost_search found a path of 10 moves: ['right', 'up', 'left', 'up', 'right', 'down', 'left', 'down', 'right', 'right']

[5차 실행]
<rand> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0528 seconds
<DFS> 이동 횟수/소요 시간: 106624 moves / Execution time: 813.9191 seconds
<BFS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0220 seconds
<A*> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0032 seconds
<UCS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0347 seconds

breadth_first_search found a path of 10 moves: ['right', 'down', 'left', 'up', 'left', 'up', 'right', 'right', 'down', 'down']
Execution time: 0.0220 seconds

aStar_search found a path of 10 moves: ['right', 'down', 'left', 'up', 'left', 'up', 'right', 'right', 'down', 'down']
Execution time: 0.0032 seconds

uniform_cost_search found a path of 10 moves: ['right', 'down', 'left', 'up', 'left', 'up', 'right', 'right', 'down', 'down']
Execution time: 0.0347 seconds

[6차 실행]
<rand> 이동 횟수/소요 시간: 22 moves / Execution time: 2.8760 seconds
<DFS> 이동 횟수/소요 시간: 60692 moves / Execution time: 166.6722 seconds

<BFS> 이동 횟수/소요 시간: 20 moves / Execution time: 1.4525 seconds
<A*> 이동 횟수/소요 시간: 20 moves / Execution time: 0.1008 seconds
<UCS> 이동 횟수/소요 시간: 20 moves / Execution time: 1.3384 seconds

breadth_first_search found a path of 20 moves: ['down', 'down', 'left', 'up', 'right', 'up', 'left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'down', 'down', 'right', 'right']

aStar_search found a path of 20 moves: ['down', 'down', 'left', 'up', 'right', 'up', 'left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'down', 'down', 'right', 'right']

uniform_cost_search found a path of 20 moves: ['down', 'down', 'left', 'up', 'right', 'up', 'left', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'up', 'left', 'down', 'down', 'right', 'right']

[7차 실행]
<rand> 이동 횟수/소요 시간: 18 moves / Execution time: 0.2182 seconds
<DFS> 이동 횟수/소요 시간: 69470 moves / Execution time: 229.8365 seconds
<BFS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0894 seconds
<A*> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0114 seconds
<UCS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0682 seconds

breadth_first_search found a path of 14 moves: ['down', 'left', 'up', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'down', 'right', 'right']

aStar_search found a path of 14 moves: ['down', 'left', 'up', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'down', 'right', 'right']

uniform_cost_search found a path of 14 moves: ['down', 'left', 'up', 'left', 'down', 'right', 'right', 'up', 'left', 'left', 'down', 'down', 'right', 'right']

[8차 실행]
<rand> 이동 횟수/소요 시간: 12 moves / Execution time: 0.2120 seconds
<DFS> 이동 횟수/소요 시간: 114714 moves / Execution time: 857.2306 seconds
<BFS> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0620 seconds
<A*> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0050 seconds
<UCS> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0436 seconds

breadth_first_search found a path of 12 moves: ['right', 'down', 'left', 'down', 'right', 'up', 'left', 'up', 'right', 'right', 'down', 'down']

aStar_search found a path of 12 moves: ['right', 'down', 'left', 'down', 'right', 'up', 'left', 'up', 'right', 'right', 'down', 'down']

uniform_cost_search found a path of 12 moves: ['right', 'down', 'left', 'down', 'right', 'up', 'left', 'up', 'right', 'right', 'down', 'down']

[9차 실행]
<rand> 이동 횟수/소요 시간: 14 moves / Execution time: 0.4661 seconds
<DFS> 이동 횟수/소요 시간: 98754 moves / Execution time: 562.4629 seconds
<BFS> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0598 seconds
<A*> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0040 seconds
<UCS> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0428 seconds

breadth_first_search found a path of 12 moves: ['up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'right', 'right', 'down']

aStar_search found a path of 12 moves: ['up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'right', 'right', 'down']

uniform_cost_search found a path of 12 moves: ['up', 'right', 'down', 'down', 'left', 'up', 'up', 'left', 'down', 'right', 'right', 'down']

[10차 실행]
<rand> 이동 횟수/소요 시간: 16 moves / Execution time: 0.2183 seconds
<DFS> 이동 횟수/소요 시간: 114772 moves / Execution time: 962.2027 seconds
<BFS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0897 seconds
<A*> 이동 횟수/소요 시간: 14 moves / Execution time: 0.0060 seconds
<UCS> 이동 횟수/소요 시간: 14 moves / Execution time: 0.1136 seconds

breadth_first_search found a path of 14 moves: ['down', 'right', 'up', 'left', 'down', 'right', 'down', 'left', 'up', 'right', 'up', 'right', 'down', 'down']

aStar_search found a path of 14 moves: ['right', 'down', 'left', 'up', 'right', 'right', 'down', 'left', 'down', 'left', 'up', 'right', 'right', 'down']

uniform_cost_search found a path of 14 moves: ['down', 'right', 'up', 'left', 'down', 'right', 'down', 'left', 'up', 'right', 'up', 'right', 'down', 'down']

코드를 실행해 보면서 BFS, A*, UCS가 같거나 비슷한 경로를 찾는 모습을 보게 되었습니다. 아래는 그 이유를 분석하여 본 것입니다.

1) BFS, A*, UCS가 같거나 비슷한 경로를 찾는 이유와 실행 시간 차이
1-1) 왜 BFS, A*, UCS의 경로는 같거나 비슷한가?
8 퍼즐 문제에서 BFS, A*, UCS가 같은 최적 경로를 찾는 이유는 세 알고리즘 모두 최단 경로를 찾는 알고리즘이기 때문입니다. BFS는 모든 간선 비용이 같을 때, 최단 경로를 찾습니다. UCS는 가중치가 존재하는 경우 최단 비용 경로를 보장합니다. 8 퍼즐에서 세 알고리즘 모두 이동 비용이 1로 같으므로, 결국 UCS는 BFS와 같은 동작을 하게 됩니다. A*는 f(n) = g(n) + h(n)을 최소화하는 경로를 찾습니다. 휴리스틱이 적절하다면 UCS와 같은 최적 경로를 찾게 됩니다. 8 퍼즐에서 세 알고리즘 모두 이동 비용이 1이므로 UCS와 BFS가 같은 방식으로 탐색하게 되고, 이렇게 하면 두 알고리즘의 최적 경로가 같게 됩니다.
A*는 휴리스틱을 추가하지만, 휴리스틱이 너무 강하지 않는 선에서 UCS와 같은 최단 경로를 찾게 됩니다. 따라서 세 알고리즘이 찾는 최적 경로가 동일 해 지는 것입니다.

1-2) 같거나 비슷한 경로를 찾지만 실행 시간이 차이가 나는 이유
같은 최단 경로를 찾더라도 탐색 방식이 다르므로 실행 시간에는 차이가 생기게 됩니다. BFS는 목표 상태에 도달하기 전까지 모든 깊이를 탐색해야 하므로 가장 오래 걸립니다. UCS는 동일 비용을 고려하지만, 정렬해야 하므로 BFS보다 약간 더 빠르거나 비슷한 성능을 보입니다. A*는 휴리스틱을 사용하여 목표 상태로 가는 경로를 우선으로 탐색하므로 불필요한 탐색을 줄여 가장 빠릅니다.

2) 코드를 실행한 결과에서 rand가 BFS, A*, UCS와 같은 경로로 8 퍼즐을 해결한 경우가 빈번하게 있었습니다. rand는 최적 경로를 찾는 알고리즘이 아닌데도 불구하고 왜 BFS, UCS, A*와 경로가 같은 때도 있는지 분석해보겠습니다.

<랜덤 탐색(rand)도 최적 경로와 같은 경우가 빈번한 이유>
2-1) 8 퍼즐의 상태 공간이 제한적이기 때문
8 퍼즐 문제는 상태 공간이 유한하며, 특정 패턴을 가지는 구조입니다. 랜덤하게 움직이더라도 특정 경로를 따라가다 보면 자연스럽게 최단 경로와 같은 경로를 선택할 확률이 높아집니다. 최적 경로는 일반적으로 10~30회 이내의 이동으로 목표 상태에 도달하게 됩니다. 랜덤 탐색이 반복적으로 실행되면서 최적 경로에 가까운 선택을 할 가능성이 커지게 됩니다. 특히 랜덤하게 선택하더라도 백트래킹을 적게 하면 최적 경로와 유사한 결과가 나오게 됩니다.

2-2) 무작위 선택이지만, 합리적인 경로를 우연히 선택한 경우
랜덤 탐색은 완전한 무작위 이동이 아니라 이동할 수 있는 방향 중 하나를 선택하는 방식입니다. 때문에 특정 패턴이 반복되면서 최적 경로를 따를 가능성이 커지게 됩니다. 예를 들어, 랜덤하게 이동한다고 해도 같은 방향으로 계속 이동하지 않고 백트래킹을 줄이는 방향을 선택하면 최적 경로를 따라갈 확률이 올라가게 됩니다. 랜덤하지만 비효율적인 탐색이 줄어들면서 최적 경로와 유사한 경로를 선택할 확률이 높아지게 되는 것입니다.

2-3) 상태 공간이 크지 않아 여러 번 실행하면 최적 경로가 나올 가능성이 큼
8 퍼즐의 전체 상태 공간은 약 181,440개로, 아주 크지는 않습니다. 랜덤 탐색을 여러 번 실행하면 최적 경로를 찾을 확률이 올라가게 됩니다. 특히 랜덤 탐색이 이미 방문한 상태를 저

장하고 있다면 불필요한 경로를 제거하여 최적 경로를 따를 확률이 더 높아지게 됩니다.
하지만 랜덤 탐색이 항상 최적 경로를 찾지는 않습니다. 운이 좋으면 최적 경로가 나오지만 아닐 경우 비효율적인 경로를 탐색할 수도 있습니다.

3. 위의 코드 실행 결과를 보면 알고리즘 중에서 A*의 실행 시간이 가장 빠른 것을 알 수 있습니다. moves = 100에서 10번 실행시켜본 결과 8 퍼즐을 푸는 데 걸리는 시간의 순위는 A* < UCS < BFS < rand < DFS입니다. A*가 8 퍼즐 문제를 다른 알고리즘보다 빠르게 푸는 이유를 분석해보겠습니다.

1) A*는 최단 경로를 찾기 위해 f(n) = g(n) + h(n)을 사용합니다.
A* 알고리즘은 실제 이동 비용인 g(n)과 추정 비용인 h(n)을 합하여 가장 비용이 적은 경로를 우선적으로 탐색합니다.

2) A*는 불필요한 탐색을 줄여서 다른 알고리즘보다 효율적입니다.
DFS는 경로가 길어질 수 있어서 매우 비효율적입니다. BFS는 최단 경로를 찾긴 하지만 모든 노드를 탐색해야 하므로 불필요한 연산이 많습니다. UCS는 최단 비용을 보장하지만, 목표 상태와 거리가 먼 노드도 고려해야 하므로 탐색 공간이 큽니다. A*는 목표에 가까운 방향으로 탐색해서 최적해를 찾으면서도 불필요한 탐색을 줄일 수 있습니다.

3) A*는 최적해를 보장하면서도 실행 속도가 빠릅니다.
A*는 UCS와 같이 최적해를 보장하면서도 휴리스틱을 사용하여 탐색 속도를 더 빠르게 개선합니다.
결론적으로 A*는 8 퍼즐과 같은 최단 경로 탐색 문제에서 가장 강력한 알고리즘이라고 볼 수가 있습니다.

4. DFS 알고리즘이 다른 알고리즘들에 비해 성능이 느린 이유/결론 (search.py 기준)
작성된 search.py를 실행시켜보면 rand의 결과가 출력되고 나서 한참 후에야 DFS의 결과가 출력됩니다. 각 알고리즘이 8 퍼즐 문제를 해결한 시간과 이동한 수를 보면 DFS는 다른 알고리즘들에 비해 확연히 좋지 않은 성능을 가지고 있다는 것을 알 수 있습니다. search.py에서 구현된 DFS 알고리즘을 기준으로 DFS가 왜 다른 알고리즘들에 비해 좋지 않은 성능을 내는지 설명하겠습니다. 우선 search.py에서 DFS는 스택을 사용하여 가장 최근에 추가된 노드부터 먼저 끝까지 탐색한 후, 더 이상 탐색할 곳이 없으면 다시 백트래킹 하는 방식으로 8 퍼즐을 해결합니다.
<DFS의 성능이 저하되는 이유>
1) 도달하고자 하는 목표 상태가 매우 깊은 그곳에 있는 경우
위에서 말했듯이 DFS는 목표 상태를 찾을 때까지 한 경로를 계속 깊게 확장하며 탐색합니다. 이런 DFS의 특성 때문에 만약 정답이 매우 깊은 위치에 있다면 DFS는 그 깊이까지 도달하는 데 오랜 시간이 걸리게 됩니다. DFS는 무작위로 깊은 노드를 탐색하기 때문에 정답을 찾기 위해 비효율적인 경로를 먼저 탐색할 가능성이 높습니다. 목표가 위치한 깊이가 깊으면 그만큼 탐색해야 하는 노드의 수도 증가하여 탐색 시간이 길어집니다.

2) 탐색 공간이 넓고 가지치기가 어려운 경우
탐색 공간이 넓을수록 DFS는 잘못된 경로를 탐색하는 시간이 증가하여 결국엔 비효율적인 결과를 얻게 됩니다. 8 퍼즐과 같은 문제에서는 각 상태에서 이동할 수 있는 경우의 수가 많으면 DFS는 비효율적인 탐색 경로를 선택할 가능성이 커지게 됩니다. 최적의 경로를 찾는 대신 잘못된 방향으로 깊이 탐색한 후 다시 돌아오는 과정이 반복되는 경우 실행 시간이 증가합니다.

3) 해답이 없는 경우(무한 탐색)
DFS는 탐색을 깊게 진행하기 때문에 문제에 대한 정답이 없는 경우 불필요한 탐색을 계속해서 반복할 가능성이 있습니다. search.py의 DFS는 순환경로를 피하기 위해 visited 집합을 사용하지만 상태 공간이 너무 크게 되면 탐색 자체가 오래 걸리게 됩니다. 특히 DFS는 깊이를 제한하지 않으면 무한 루프에 빠질 위험이 있습니다.

4) 최적해를 보장하지 않습니다. (비효율적인 경로 선택)
DFS는 최단 경로를 보장하지 않습니다. 문제에 대한 정답이 어디에 있느냐에 따라 긴 경로를 탐색할 수도 있습니다. DFS는 목표 상태를 찾는 즉시 반환하기 때문에 더 짧은 경로를 찾을 기회를 놓칠 수 있습니다.

search.py에서 DFS는 스택으로 노드들을 깊이 우선 탐색합니다. 또 visited 집합을 사용하여 한번 방문한 노드의 재방문을 방지합니다. 하지만 가지치기하지 않고 깊이 탐색하는 DFS의 특성으로 인해 경로가 비효율적이거나 불필요하게 깊이 탐색하는 문제점이 발생합니다. BFS, UCS, A*와 비교하였을 때 DFS는 목표 상태를 찾는 속도가 느리고 비효율적인 경로를 선택하게 됩니다. DFS는 탐색 공간이 작고 목표 상태가 깊지 않을 때 유용하게 사용할 수 있지만 일반적으로 8 퍼즐 문제에서는 BFS, UCS, A*가 더 효율적인 방법입니다.

5. moves가 성능에 영향을 주는가?
랜덤 퍼즐을 생성하기 위해 원래 목표 상태에서 moves의 수만큼 퍼즐을 섞은 다음 알고리즘들로 퍼즐을 풀게 됩니다. 그 과정에서 퍼즐을 랜덤하게 섞는 moves의 횟수에 따라 각 알고리즘이 8 퍼즐을 해결하는 데 영향을 받는지 받지 않는지를 알아보기 위해
moves를 10, 50, 150, 300으로 놓고 코드를 실행시켜보았습니다.

moves = 10일 때
<rand> 이동 횟수/소요 시간: 2 moves / Execution time: 0.0010 seconds
<DFS> 이동 횟수/소요 시간: 2 moves / Execution time: 0.0000 seconds
<BFS> 이동 횟수/소요 시간: 2 moves / Execution time: 0.0000 seconds
<A*> 이동 횟수/소요 시간: 2 moves / Execution time: 0.0000 seconds
<UCS> 이동 횟수/소요 시간: 2 moves / Execution time: 0.0010 seconds

moves = 50일 때
<rand> 이동 횟수/소요 시간: 16 moves / Execution time: 2.3263 seconds
<DFS> 이동 횟수/소요 시간: 45394 moves / Execution time: 82.2886 seconds

<BFS> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0508 seconds
<A*> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0040 seconds
<UCS> 이동 횟수/소요 시간: 12 moves / Execution time: 0.0518 seconds
moves = 150일 때
<rand> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0189 seconds
<DFS> 이동 횟수/소요 시간: 24698 moves / Execution time: 16.0450 seconds
<BFS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0097 seconds
<A*> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0010 seconds
<UCS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0104 seconds

moves = 300일 때
<rand> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0040 seconds
<DFS> 이동 횟수/소요 시간: 31908 moves / Execution time: 35.3890 seconds
<BFS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0179 seconds
<A*> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0020 seconds
<UCS> 이동 횟수/소요 시간: 10 moves / Execution time: 0.0130 seconds

위의 결과에서 moves가 10과 같이 작을 때에는 문제 난이도가 낮아서 탐색 공간이 작고, 알고리즘들이 빠르게 답을 찾을 수 있습니다. 특히 DFS, A*, UCS는 거의 차이가 없습니다. 하지만 반대로 moves가 50 이상 커질 때는 상태 공간이 일정 수준 이상 커지면 DFS, A*, UCS는 거의 비슷한 실행 시간을 보이게 됩니다. DFS는 여전히 비효율적이지만, 실행 시간 증가 폭이 점점 둔화하는 모습을 확인할 수 있었습니다. 따라서 moves가 작을 때에는 알고리즘들의 문제를 해결하는 속도가 빨라지지만, 50 이상으로 커질 때는 moves의 수가 커짐에 따라 알고리즘이 문제 해결 속도가 느려지는 것이 아니라 어떻게 섞였느냐에 따라 그 속도가 빨라지거나 느려지는 것이라고 생각합니다.