8-puzzle을 해결하는 탐색 알고리즘 구현 및 결과 보고서

202312632 김예은

초기 코드에서는 경로를 표현할 때 노드 리스트를 사용하는 구조로 되어 있으며, 모든 상태를 저장하도록 구현되어 있기 때문에 탐색 중인 노드 하나를 위해 경로 전체를 복사하여 저장한다는 문제점이 있다고 판단하였습니다. 예를 들어서 DFS에서 깊이가 1000 이상이 되면 1000개 리스트가 매번 복사되기 때문에 탐색 깊이가 깊어질수록 성능 저하와 메모리 낭비가 발생하는 문제가 생겨, 결과 출력까지 걸리는 시간이 너무 낭비되는 느낌을 받았습니다. 따라서 다음과 같이 코드를 수정하였습니다.

node 클래스를 도입하여서 state, action, cost, parent 의 정보를 하나의 객체에 캡슐화했습니다. path() 메서드는 현재 상태의 노드부터 루트 노드인 시작 상태까지의 경로에서 어떠한 action(up, down, left, right)들을 했는지 parent를 따라 거슬러 가면서 action을 리스트에 저장하고 reverse()를 취하여 올바른 순서대로 반환하도록 구현하였습니다. 즉, action 리스트를 반환하는 역할을 합니다. 각각의 탐색 알고리즘들에서 동일한 node 클래스를 사용하도록 설계하여 알고리즘 간 구조를 통일함으로써 효율성을 높이는 데 초점을 맞췄습니다.

DFS는 깊이 우선 탐색이기 때문에 스택 기반으로 last in, first out 구조를 사용합니다. node(start)를 스택에 넣고, 가장 나중에 들어온 노드를 pop() 하여서 탐색합니다. 목표에 도달하면 node.path()로 경로를 반환합니다. node.parent 를 통해 경로를 기억합니다.

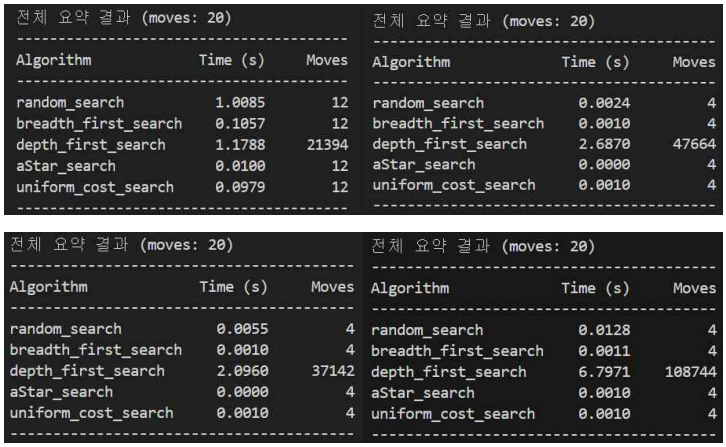
BFS는 너비 우선 탐색이기 때문에 first in, first out 큐를 사용합니다. node(start)를 큐에 넣고, 가장 먼저 들어온 노드를 popleft() 하여서 탐색합니다. 자식은 새로운 노드로 큐에 넣습니다. node.path()로 도달 경로를 반환합니다.

UCS는 비용을 기준으로 합니다. 가장 비용이 작은 노드부터 확장합니다. 시작 노드를 (0, node(start)) 형태로 우선순위 큐에 넣고 heapq에서 가장 비용이 작은 노드를 꺼내고, 자식 상태를 누적 비용을 포함한 노드로 생성합니다. 비용은 힙 정렬되고 __lt__() 메서드로 비교합니다.

A*는 UCS와 동일하게 비용 기준으로, 시작 상태를 큐에 저장합니다. heapq에서 f(n)이 가장 작은 노드를 꺼냅니다. 힙 정렬할 때 튜플의 첫 번째 요소가 되는 f 값을 기준으로 해서 우선순위 정렬합니다. 휴리스틱은 노드 내부에는 없으나 외부에서 계산된 후 전달됩니다.

다양한 결과들을 분석하기 위해 main.py을 수정하여 각 알고리즘들의 난이도 설정, 즉, 퍼즐을 섞는 횟수를 다르게 설정함에 따라 시간이 얼마나 걸리는지, 총 움직인 횟수는 몇 번인지 결과를 요약해서 출력하도록 설계하였습니다. moves를 20(비교적 쉬운 난이도), 30(중간 난이도), 50(어려운 난이도)로 설정한 후 분석한 결과는 다음과 같습니다. createRandomEigthPuzzle()이 매 실행마다 새로운 다른 퍼즐들을 생성하기 때문에 이에 따라 같은 moves일지라도 결과에 편차가 존재했습니다. 따라서 같은 퍼즐에 대해서 여러 번 실행함으로 평균을 확인하여 분석하는 방식을 선택하였습니다.

moves = 20 →



전체 요약 결과 (moves: 20)

Algorithm	Time (s)	Moves
random_search	1.0085	12
breadth_first_search	0.1057	12
depth_first_search	1.1788	21394
aStar_search	0.0100	12
uniform_cost_search	0.0979	12

전체 요약 결과 (moves: 20)

Algorithm	Time (s)	Moves
random_search	0.0024	4
breadth_first_search	0.0010	4
depth_first_search	2.6870	47664
aStar_search	0.0000	4
uniform_cost_search	0.0010	4

전체 요약 결과 (moves: 20)

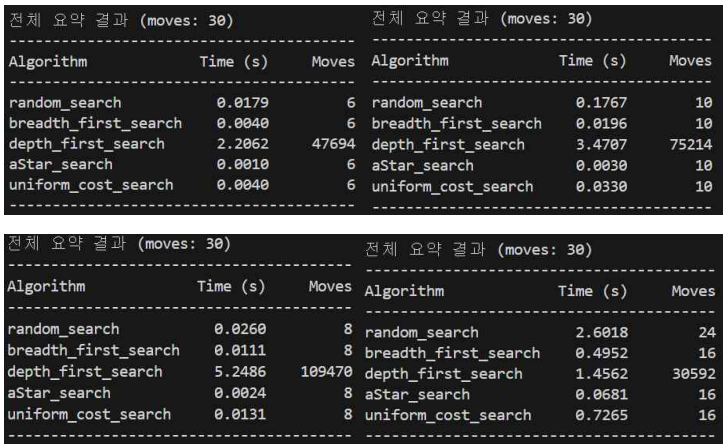
Algorithm	Time (s)	Moves
random_search	0.0055	4
breadth_first_search	0.0010	4
depth_first_search	2.0960	37142
aStar_search	0.0000	4
uniform_cost_search	0.0010	4

전체 요약 결과 (moves: 20)

Algorithm	Time (s)	Moves
random_search	0.0128	4
breadth_first_search	0.0011	4
depth_first_search	6.7971	108744
aStar_search	0.0010	4
uniform_cost_search	0.0010	4

---

moves = 30 →



전체 요약 결과 (moves: 30)

Algorithm	Time (s)	Moves
random_search	0.0179	6
breadth_first_search	0.0040	6
depth_first_search	2.2062	47694
aStar_search	0.0010	6
uniform_cost_search	0.0040	6

전체 요약 결과 (moves: 30)

Algorithm	Time (s)	Moves
random_search	0.1767	10
breadth_first_search	0.0196	10
depth_first_search	3.4707	75214
aStar_search	0.0030	10
uniform_cost_search	0.0330	10

전체 요약 결과 (moves: 30)

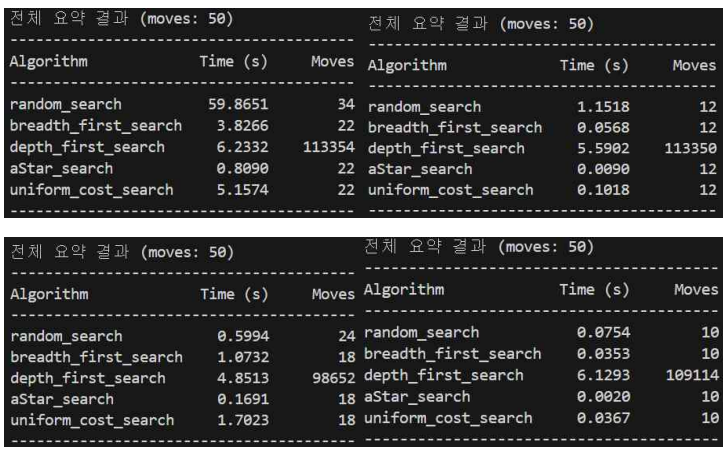
Algorithm	Time (s)	Moves
random_search	0.0260	8
breadth_first_search	0.0111	8
depth_first_search	5.2486	109470
aStar_search	0.0024	8
uniform_cost_search	0.0131	8

전체 요약 결과 (moves: 30)

Algorithm	Time (s)	Moves
random_search	2.6018	24
breadth_first_search	0.4952	16
depth_first_search	1.4562	30592
aStar_search	0.0681	16
uniform_cost_search	0.7265	16

---

moves = 50 →



전체 요약 결과 (moves: 50)

Algorithm	Time (s)	Moves
random_search	59.8651	34
breadth_first_search	3.8266	22
depth_first_search	6.2332	113354
aStar_search	0.8090	22
uniform_cost_search	5.1574	22

전체 요약 결과 (moves: 50)

Algorithm	Time (s)	Moves
random_search	1.1518	12
breadth_first_search	0.0568	12
depth_first_search	5.5902	113350
aStar_search	0.0090	12
uniform_cost_search	0.1018	12

전체 요약 결과 (moves: 50)

Algorithm	Time (s)	Moves
random_search	0.5994	24
breadth_first_search	1.0732	18
depth_first_search	4.8513	98652
aStar_search	0.1691	18
uniform_cost_search	1.7023	18

전체 요약 결과 (moves: 50)

Algorithm	Time (s)	Moves
random_search	0.0754	10
breadth_first_search	0.0353	10
depth_first_search	6.1293	109114
aStar_search	0.0020	10
uniform_cost_search	0.0367	10

• Random Search는 moves를 높게 설정할수록 시간과 이동 횟수에 있어서 차이가 심합니다. 자식 노드를 무작위로 선택해 탐색하기 때문에 단순하고 비효율적입니다. 동일한 난이도로 테스트를 진행했음에도 최악의 경우에는 59초, 최고는 1초로 편차가 극심하다는 것을 확인할 수 있었습니다.

• DFS는 20, 30, 50의 모든 상황에서 항상 많은 노드를 탐색합니다. moves가 거의 항상

수만~수십만 회까지 이르고 시간도 급격하게 증가하는 것을 볼 수 있습니다.

• DFS는 위에서 언급했던 것처럼 깊이 우선 탐색이기 때문에 해가 깊은 곳에 있을 경우에는 빠르게 도달할 수 있다는 장점이 있지만, 그에 반해 해가 만약 얕은 깊이에 있다고 하더라도 불필요하게 깊이 우선 탐색을 진행하므로 문제가 된다고 분석하였습니다. 예를 들어서 5 moves만에 도달이 가능함에도 DFS는 깊이 우선 탐색을 하다가 막히면 이전 분기점으로 돌아가서 다시 탐색을 하므로 시간이 오래 걸릴 수밖에 없습니다. 결과적으로 해에 도달할 때까지 매우 많은 노드를 확장하게 되고, 그 과정에서 시간과 메모리 상에 모두 비효율적이게 됩니다. 따라서 실제로 직접 실행을 해보았을 때에도 moves가 수만~수십만까지도 찍혔던 것을 확인할 수 있었습니다. 문제를 효율적으로 해결하는 느낌이 아니라 끝까지 가서 해를 찾으며 돌아오는 느낌으로 매우 비효율적인 방식이라고 생각됩니다.

• BFS, A*, UCS는 Random Search와 DFS에 비해서 훨씬 안정적이면서 효율적입니다.

• 특히 A*는 모든 결과에서 가장 빠른 경로를 탐색함을 보여줍니다. 모든 알고리즘들을 종합해 봐도 항상 빠르게 최적의 해를 찾고, 실행 시간과 이동 횟수 면에서 가장 우수한 성능을 보였습니다.

• 난이도가 높아짐에 따른 차이를 확인해본다면, A*는 꾸준하게 좋은 성능을 보입니다. 그에 반해 DFS는 난이도가 증가하면 시간과 이동 횟수에 있어서 아주 취약하고 비효율적인 지표를 보여줍니다.

• DFS는 스택을 사용하기 때문에 이론적으로 공간 복잡도가 O(bm)입니다. 이론상 메모리는 BFS보다 적게 쓰는 것이 맞지만, 실험 결과에서는 오히려 많은 시간을 소모하는 모습을 확인할 수 있었습니다. 따라서 나머지 알고리즘들에 비해서 실용성이 현저히 떨어집니다.

• BFS와 UCS는 각각 큐 또는 우선순위 큐를 사용해서 상태들을 저장합니다. 이때 큐의 크기가 공간 복잡도, 알고리즘이 실행될 때 필요한 메모리 양에 영향을 줍니다. 이론적으로 BFS의 공간 복잡도는 O(b^d)입니다. 모든 깊이 d까지의 노드를 저장하기 때문에 큐가 매우 커지게 됩니다. UCS는 BFS와 비슷한데, 비용 순서로 큐에서 꺼내집니다. heapq로 구현된 우선순위 큐를 사용하는 거시 일반적입니다. 이론적으로 UCS의 공간 복잡도는 O(b^c)입니다. BFS와 유사하지만 이때 c는 최적해의 누적 비용이고, 더 많은 상태를 중복 저장할 수가 있습니다. 따라서 BFS보다 시간면에 있어서는 효율적이지만 메모리로 따진다면 더 많이 사용할 수 있습니다. 두 알고리즘 모두 상태를 큐에 저장하기 때문에 공간 복잡도가 높습니다. 하지만 최단 경로는 항상 보장된다고 분석하였습니다.