인지과 2주차 - 2. 탐색

역사, 정의, 활용 (전반적 내용) + 어떤 것 배울지? → 1주차

상상한 AI ≠ 학문적 AI (수업내용) 갭 있음 → 학문 이해 시간, 연구 多 필요

"개론" → 전반적인 이야기 / 실제 적용 & 연구적 내용은 딥러닝, 기계학습으로 ??

컴퓨터 개발 + 기대↑ ⇒ "컴퓨터가 사람 대신할 것" → 인공지능!

"탐색으로부터 시작" → 문제가 주어졌을 때 문제 해결을 위한 과정.

얼마나 컴퓨터가 효율적으로 사람처럼 둘 수 있을까?

01. 탐색

알파고 : 딥러닝 + 몬테카를로 트리

→ 탐색의 대표주자

→ 초기 상태 + 목표 상태 정해야 함.

바둑은 경우의 수 매우 多. ⇒ 상대방을 이기기 위해 탐색하면 매우 많은 시간 소요.

아주 간단한 게임은 쉬운 탐색, but 체스 · 바둑같은 탐색 공간 큰 경우는 hardware 뒷받침 필요.

→ 과거엔 X. 2000년대 이후 hardware 발전 영향받음. 탐색 시간 ↓.

문제: 그래도 경우의 수 많음 (바둑). ∴ 제한된 시간 내에 어떻게 탐색 마치고 결과 도출?

현재 컴퓨터로는 실제로 전체 다 보고 결정 불가. ⇒ 몬테카를로 트리라는 탐색 알고리즘 활용.

몬테카를로 트리 : 이길 수 있는 수만 추려서 몇 개의 가능성 있는 것만 탐색

이세돌의 "신의 한 수" ⇒ 알파고 입장에선 처음 보고 예측 불가능했던 수. 학습하지 않은 부분.

이길 수 있는 경우의 수만 탐색했기 때문.

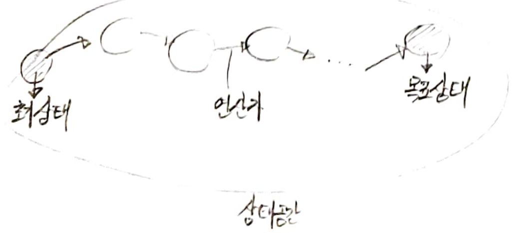
결국, 탐색은 초기 상태에서 목표 상태까지 갈 수 있는 여러 경우의 수를 두고 효율적으로 목표상태에 도달 가능한 경로를 찾는 것.

이겨야 하는 확률을 걸고 진행, 확률이 높은 경우의 수를 선택하는 방식.

Why 확률? : 내가 하려는 행동은 상대방 행동에 영향받음. ∴ 무조건 이긴다고 볼 수 X (언제는 내가 높다/낮다)

탐색 정의 : 상태공간에서 시작상태에서 목표상태까지의 경로를 찾는 것

경우의 수를 모아놓은 공간 / ⊂상태공간 문제 해결을 시작하는 단계. / ⊂상태공간 목표가 되는 상태.



연산자: 한 상태에서 다른 상태로 이동하기 위해 하는 것.

상태를 변경시키는 모든 함수. (혹은) 다른 상태를 생성하는 것.

ex) 시계방향, 수를 두는 것, 퍼즐 조각을 두는 것 등의 모든 함수

행동 · 함수 통해 다른 상태로 이동.

Why 생성? → 모든 상태를 상태공간 안에 표현하면 저장공간(메모리)의 문제 발생.

도형 분기 확장 시 시간 · 메모리 절약.

02. 상태 공간의 예.

초기 인공지능 → 탐색 연구 시에 복잡한 문제 해결 X. 간단한 8-퍼즐 가지고 연구 시작.

8-퍼즐 : 초기 상태, 목표 상태, 연산자* 정의 필요.

→ 8개의 숫자가 적힌 3×3 퍼즐판을 순서대로 정렬 필요

	1	3
4	2	5
7	8	6

초기 상태

>>>>

1	2	3
4	5	6
7	8	

목표 상태

상태 공간

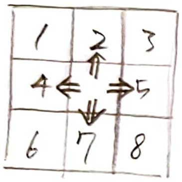
전체 경우의 수 : 9! (그런 하기만 계산 가능)

∴ 초기 → 목표로 가는 경우의 수 ∞

근데 연산자를 어떻게 정의하는가? 가 문제가 됨.

⇒ 빈칸이 움직이는 방향으로 정의. 숫자가 아니라! + 상하좌우로 숫자로 하면 연산자로 인한 연산이 복잡해짐.

연산자



⇑ : UP

⇐ : LEFT

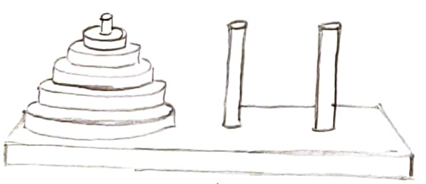
⇒ : RIGHT

⇓ : DOWN

실제 컴퓨터를 통해 연산을 수행할 때 : 타일 순서 주어져야 함 + 빈칸 이동해야 함 (4가지 연산 이동)

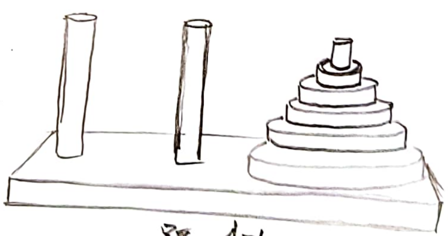
but 실제로 4가지 연산을 모두 적용하는 case는 빈 칸이 한가운데일 뿐. 다른 곳은 위치의 제약.

하노이 탑



초기 상태
(순서대로 정리된 원반들이 맨 왼쪽에 존재)

⟹



목표 상태
(순서대로 맨 오른쪽으로 가서 존재함)

조건 : 큰 원반이 작은 원반 위에 올라갈 수 X + 한번에 하나의 원반만 이동 가능

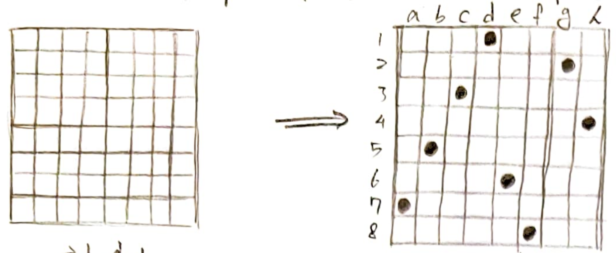
경우의 수? ⇒ 2개 원반이라 했을 경우 3번의 이동이 최적화.
3개 원반일 경우 7번의 이동이 최적화.
} ⇒ $2^n - 1$회 ∴ 원반이 많아질 수록 기하급수적으로 시간 매우 ↑.

연산자? ⇒ 원반을 이동 (어떤 원반 + 어디로 이동)

- State : $A = \{(a_1, a_2, a_3) \mid a_i \in \{A, B, C\}\}$ → 3개의 원반이라 했을 때. a_i : 원반
- 초기 상태 : $I = (A, A, A)$
- 목표 상태 : $G = (C, C, C)$
- 연산자 : $O = \{Move_{which,\ where} \mid which \in \{1, 2, 3\},\ where \in \{A, B, C\}\}$ → 어떤 원반을 어떤 기둥으로?

8-Queens : 8×8 체스판에 2개의 퀸이 서로를 위협하지 않도록 8개의 퀸을 배치하지 않는 문제

⇒ 두 개의 퀸이 같은 행, 열 또는 대각선을 공유하지 않도록 하는 문제. 위협을 check 해야 함



초기 상태 (퀸 하나도 ×) ⟶ 목표 상태 (조건 만족하며 배치 완료 ; 위는 목표 상태 中 1개)

연산자? ⇒ 퀸을 놓는 것 ! ; $O = \{Place_i \mid 1 \le i \le 8\}$ (여기선 행을 정하는 것.)

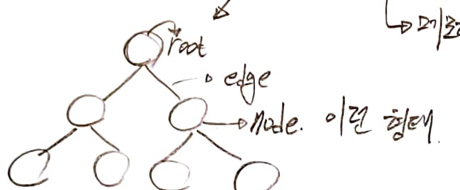
왜 행/열 둘 관계인 고른다? ⇒ 행 위치가 정해지면, 열은 규칙 위배되는 곳은 제외한 나머지 중에서 자동 선택. 어떤 행에 놓더라도 진행이 안되면, 자동으로 올라감.

03. 탐색 트리

탐색 트리 : 하나의 상태에서 연산자를 적용하면 생성되는 새로운 상태들이 있는데, 이를 트리 구조로 나타내 정리해 보기 만든 것.

⇒ 어떻게 진행될지를 예측, 진행 상황을 알 수 있음.

→ 초기 상태에서 연산자를 적용하면 생성되는 트리 구조 ; 모든 노드가 미리 생성되어 있지 않음 → 메모리 상 비효율적



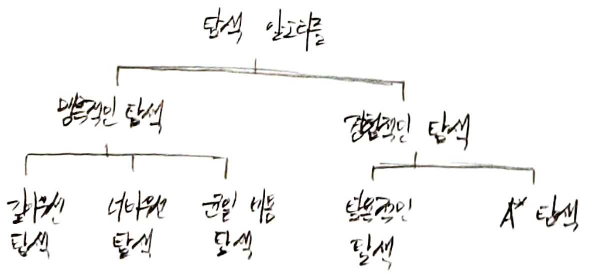
상태 : Node   초기 상태 : Root Node   연산자 : Edge

확장 (expand) : 어떤 노드에 연산자를 적용하여 새로운 상태를 생성하는 것.

(Open : 아직 확장되지 않은 노드들.
(Close : 확장이 끝난 노드들

→ Why? If 연산자 적용하여 새 상태 생성되면 여러가지 연산 적용해 확장 될 수 있음. 근데 확장 가능한 경우의 수 중에 하나를 택했는데 목표 상태로 가지 않는 것 같으면 돌아와서 적용 가능한 다른 연산자 적용해야 함. 이 때 적용 했던 것들은 구분이 필요함. 안 할 경우 탐색을 무한히 반복해야 함.

04. 기본적인 탐색 기법.



맹목적 탐색 : 목표 노드에 대한 정보 없이 기계적인 순서로 노드를 확장하는 방법.

→ 어떤 정보를 가지고 목표 상태에 도달하는 지는 관심 ×. 오로지 확장 우선.

→ 소모적 탐색. 기계적임 "비효율적"

경험적 탐색 : 목표 노드에 대한 경험적인 정보 사용

→ 문제 해결하는 데 경험을 사용 ; 효율적 탐색

맹목적 { 장점 : 100% 확률로 목표 상태에 도달함.
　　　 { 단점 : 비효율적

경험적 { 장점 : 효율적임
　　　 { 단점 : 상태 공간이 커진다면 경험의 영향력이 줄어듦. → 반비례.
　　　　 → 목표 상태 도달 확률 줄어듦. 모든 경험이 옳은 경로는 ×.

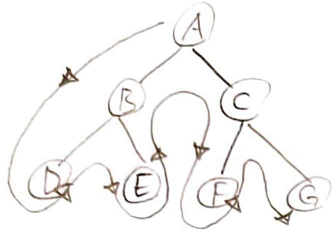
05. 깊이 우선 탐색 (DFS)

DFS: 탐색 트리 상에서, 해가 존재할 가능성이 존재하는 한, 앞으로 (밑으로 ; 깊이) 계속 전진하며 탐색하는 방법.

→ 목표 노드가 깊은 곳에 있는 경우 빠르게 해를 구함. but 해가 없는 경로에 빠지면 나오지 못할 수 있음.

트리의 깊이가 충분히 깊다면... → 계속 내려가기만 함. (해가 나온다는 장담을 못하는 상태에서!)

→ 합리적 시간 안에 되돌아오고, 유의미한 결과를 주기 위해 트리의 깊이를 제한하는 경우도 있음



ABDECFG 순서 탐색

ex) D.E 쪽에 해가 있다면 절반만 탐색했으므로 빠른 결과.
F.G 쪽에 해가 있다면 B,D,E를 탐색하므로 (관계X인 노드) 느림 (상대적으로)

8-퍼즐에서의 DFS 예

초기 상태: 빈 공간이 가운데 ⇒ 이 노드 A에서 4개의 연산자를 적용 노드 B, C, D, E 생성

DFS시 노드 B 선택 (up 연산자 우선 적용)

부모 노드와 중복되지 않는 연산 수행을 통해 노드 확장

상태 바꾼 후 check {
1. 목표 노드인가?
2. 부모 노드와 같은가? → 같다면 탐색 필요X. 무한반복이니까.

ex) up 후 down 하면 다시 root 노드임

중복 상태? → 연산자 적용 시 부모노드와 동일해지는 상태.

→ 중복 상태를 막기 위해 2개의 리스트 사용. {
open: 확장되었으나 아직 탐색 안된 상태들이 있는 리스트
Closed: 탐색 끝난 상태들이 들어있는 리스트

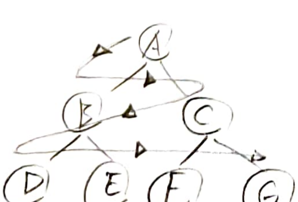
구현하지 않으면 반복이 지속될 수

DFS 알고리즘 PseudoCode → 강의 교안 확인. (핵심: 스택자료 사용)

06. 너비우선 탐색 (BFS)

BFS: 루트노드의 모든 자식 노드들을 탐색한 후에 해가 발견되지 않으면 한 레벨 내려가서 동일한 방법으로 탐색을 계속하는 방법.

→ 알고리즘 PseudoCode → 강의 교안 확인. (핵심: 큐자료 사용)



ABCDEFG 순서 탐색

BFS, DFS 모두 알고리즘의 효율성 면에서 깊은 목표 노드가 깊은지 / 넓은지 등 상황에 따른 효율성은 존재하나, 알고리즘의 전체적인 부분을 봤을 땐 효율 큰 차이 존재X.

균일 비용 탐색은 알아서 찾아보길.

DFS, BFS 배우며 체크할 부분 : open, closed 리스트와 각 상태에서의, 즉, 초기에서 목표 상태로 가는 과정의 임의의 상태에서의 open에 있는 노드들의 순서와 closed에 있는 상태들의 순서 등
→ 어떤 것이 open, 어떤 것이 closed에 있는지 확인

07. 실습

InJiGoe_W2_02.py 참고
교재 pg. 76 ~ 80

08. 경험적 탐색 방법

문제 영역에 대한 정보/지식을 사용할 수 있다면 탐색 작업을 훨씬 빠르게 진행
→ 경험적 탐색 방법 (heuristic search method) 라고 함.
이 때 사용되는 정보를 휴리스틱 정보라 함.

임의의 상태까지 왔을 때 인간이라면 "어떻게 하면 답이 나오겠다 / 안나오겠다"를 경험적으로 판단 가능.
→ 이 경험을 컴퓨터에 알릴 방법이 필요. 그것이 쉽게 X. → 정성적이기보다 정량적 방법을 알려주는 것이 더 효율적
평가함수 : 현재 상태에서 경험치를 계산하고 반환하는 함수. 계산 가능한 함수 식을 만들어줌 → 최대한 간단하게
→ 휴리스틱 값.
Why? → 목표와 가까운 노드들과 목표와 먼 노드 간에는 값의 차이를 분명히 볼 수 있음.
이를 잘만 만들면 경험을 정량적 표현으로 바꿀 수 있음.

8-퍼즐에서의 휴리스틱 정보

$h_1(N)$: 제 위치에 있지 않은 타일의 개수. → 있지 않으면 1로 해서 1의 값이 많을수록 목표 상태와 거리가 멂.
$h_2(N)$: 각 타일의 목표 위치까지의 거리 → 합이 클수록 목표 상태와 거리 ↑.

09. 언덕 등반 기법 (Hill-Climbing)

앞에서 쉽든 평가 함수를 만들었는데, 각 상태에 대한 평가 함수 계산 가능.
→ 탐색 트리 만들고, 탐색 트리 이용해 탐색 시에 기준이 있었는데 (맹목적 탐색), 경험적 탐색은 평가함수 값을 기준으로 함.
↳ 깊이 OR 너비 순서로; 바른지는 모름. ↳ 작은 값부터.
→ 평가 함수의 값을 이용하여 바른 탐색을 시도.
언덕 등반 기법 : 경험적 탐색의 한 방법; 평가 함수 값이 좋은 노드를 먼저 처리

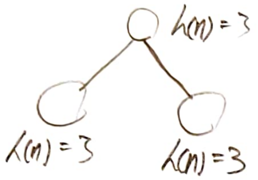
경험적 탐색 방법은 휴리스틱 함수 값이 가장 좋은 노드만 선택!
↓
알고리즘 :
1. 현재 위치를 기준으로 각 방향 높이 확인 (노드 확장)
2. 모든 위치가 현 위치보다 낮다면 그 곳을 정상이라 판단함 (목표상태인가 검사)
3. 현 위치가 정상이 아니라면 확인된 위치 중 가장 높은 곳으로 이동 (후계 노드 선택)

지역 최소 문제: 순수 언덕 등반 기법은 오직 $h(n)$ 값만을 사용. (open/closed 리스트 사용 x)

→ 생성된 자식 노드의 평가함수 값이 부모 노드보다 더 높거나 같은 경우가 나올 수 있음. ; 지역 최소 문제라 함.
local minima problem

→ 낮은 쪽으로 이동해야 하는데 모든 자식 노드가 부모 노드보다 높으면 선택할 것이 X.

예시: 부모 노드 $h(n)$ 값 == 자식 노드 $h(n)$ 값 → 어떤 것 선택? "불분명함"



→ 무엇이든 하나를 정하도록 알고리즘 설계해야.

최고 우선 탐색 : 언덕 등반 기법 개선. ; 평가 함수 활용 + open/closed 리스트 병행 활용.

→ Open 리스트 중 평가 함수 값이 가장 좋은 노드 선택

- 언덕 등반 기법은 생성된 자식 노드들의 저장이 없이 한번 선택되지 못하면 다시 선택할 기회 X.
- 최고 우선 탐색은 open 리스트에 저장된 자식 노드들에게 기회 주어짐.

11. A* 알고리즘

A* 알고리즘 ; 평가 알고리즘 → $f(n) = g(n) + h(n)$ 으로 정의 (n: 경로 상의 노드)

$h(n)$: 현재 노드에서 목표 노드까지의 거리

$g(n)$: 시작 노드에서 현재 노드까지의 비용. ; 레벨?

→ 최소 비용 경로 반환 보장 알고리즘

어떤 경로 선택할 지는 평가함수를 기준으로, 목표 상태에 도달했는지는 초기 상태와 목표 상태에 도달한 거를 비교.

경험적 알고리즘이 효율적으로 보임. but 효율적이기 위해선 각 평가함수를 잘 결정해야 함.

→ 일관성 X OR 올바르지 않으면 오히려 독이 됨.

A* 알고리즘 pseudocode → 강의 교안 확인.