인지개 3주차 — 3. 게임트리

2주차: 탐색 → 상태공간 정의(+초기 상태, 목표 상태, 연산자)를 통한 최적 경로 탐색 과정

01. 게임 프로그램

전제: 두 명의 경기자 & 제로섬 게임 & 차례대로 수를 두는 게임
- 제로섬 게임 ↳ 누군가 이기면 누군가는 짐
- 차례대로 수를 두는 게임 ↳ 턴제로 번갈아 함

→ ex) 체스, 바둑, 오목, ...

→ 적은 수의 연산자 정의되며 연산 결과는 규칙으로 정의됨 ; 일관적으로 연산자 사용 가능

게임 정의: 2인용 게임 & 두 경기자를 Max, Min으로 정의 → 항상 Max가 먼저 수를 두는 것으로 가정

Why? 나를 Max, 컴퓨터를 Min으로 하면 나는 이길 확률이 높도록 트리를 확장하려 할 것이고, 상대(컴퓨터)는 나의 이길 확률이 낮아지도록 트리를 확장하려 할 것임 → 둘 다 영향을 미침
나도 잘하지만 상대방도 잘한다는 가정 하에 이뤄짐 ; 어떻게 하면 내가 이기는 경로를 찾을 수 있는가?

Tic-Tac-Toe 게임트리: 3×3 게임보드 (9! 경우의 수) ; 단말 노드(↳ 게임 종결 상태) 평가값 0, -1, +1
- 0 → 비김
- -1 → Min 승
- +1 → Max 승

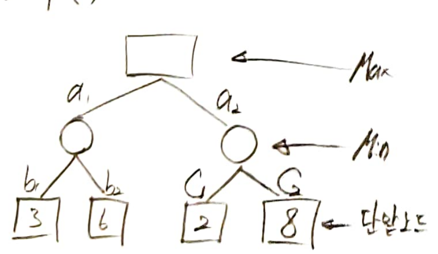
→ 가로, 세로, 대각선 3개가 연속으로 X 또는 O이면 게임 종료

02. 미니맥스 알고리즘

게임 트리에서의 목표 상태: 승리에 해당하는 단말 노드의 상태 ; 게임은 상대방이 탐색에 영향을 미침
↳ 1개가 아니라 여러 개이다

→ 상대방이 최선의 수를 둔다고 가정하여 탐색을 진행 → 즉, 상대방이 최선의 수를 두지 않으면 쉬운 탐색이 가능하다!

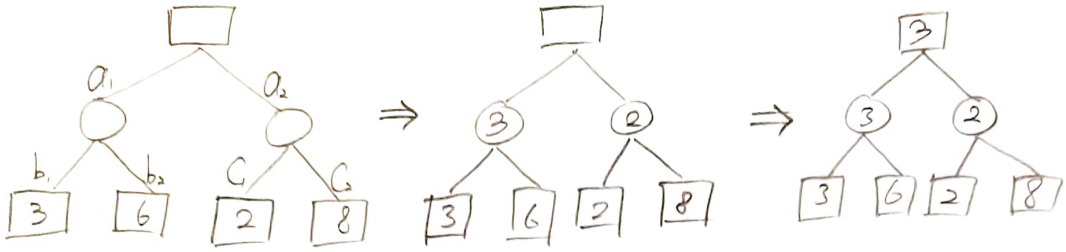
게임 예 (1)



- Max: a_1, a_2 두 개의 상태 생성
- Min: a_1 상태에서 b_1, b_2 & a_2 상태에서 c_1, c_2 상태 생성
- 게임 평가값: 2~8, 높을수록 Max가 유리

→ 트리만 봤을 땐 Max는 a_2 → 8 루트를 선택할 것. 그러면 Max만 게임하는 중.
Min도 게임을 하기 때문에 Max가 최고의 값을 갖지 못하게 최소의 수를 두려고 할 것.
∴ 항상 최선을 목표로 하나 차선책을 통해 이기는 수를 찾아가는 것이 미니맥스 알고리즘

게임 예 (2)



미니맥스 알고리즘 : 전체의 단말노드가 정해져 종료된 후 역으로 거슬러 선택

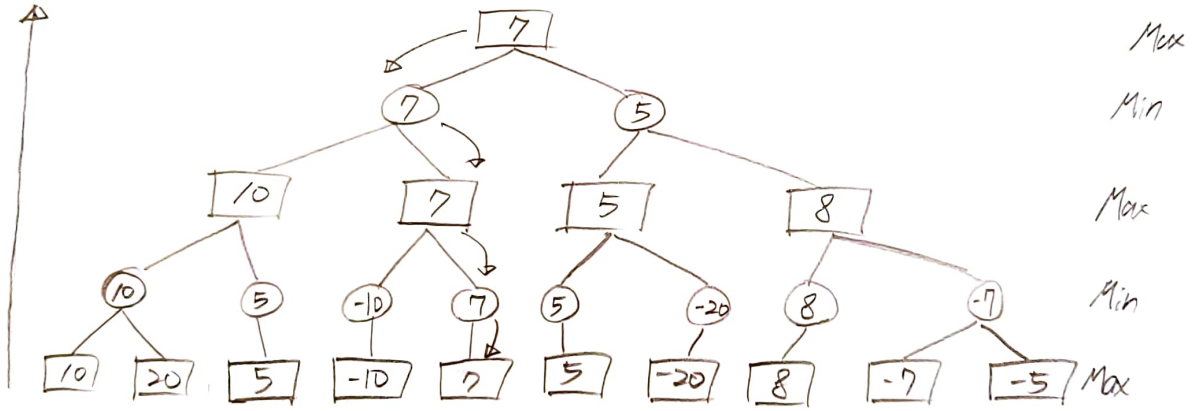
Min: ↓값 유리 → b_1 vs b_2 에서 b_1, C vs C 에서 C 선택

Max: a_1 vs a_2 中 높은 값 선택해야 되므로 a_1 선택

Tic-Tac-Toe에 미니맥스 알고리즘 적용

→ 평가함수: Max(X)가 이기면 +1, Min(O)가 이기면 -1, 비기면 0
같의 코인 참고

미니맥스 알고리즘 실습



역으로 올라가면서 구함

미니맥스 알고리즘의 시간 복잡도

→ 미니맥스 알고리즘은 DFS를 수행 ; 깊이 m이고 각 노드별 가능한 경우의 수가 b인 경우 $O(b^m)$

바둑의 경우 → 경우의 수 $3^{16}! = 10^{761}$; 어마어마한 시간 복잡도

→ 이 때문에 기존엔 간단한 알고리즘만 해결 가능했음

경우의 수 多 경우 → 크게 수익을 얻을 목표 ; 이 때 뭘 탐색하는 알고리즘의 포인트? → "어떻게 평가함수를 잘 쓰나"
but if 평가 시에 나온 평가값이 전부 동일하다면? (가령, 깊이가 얕으면 살짝화는 깊이 중요함) 문제가 될 수도
→ ∴ 평가 함수를 깊이 맞게 잘 설정해야 함

03 코드실습

InJiGae_113_02.py 참고

04. 알파베타 가지치기

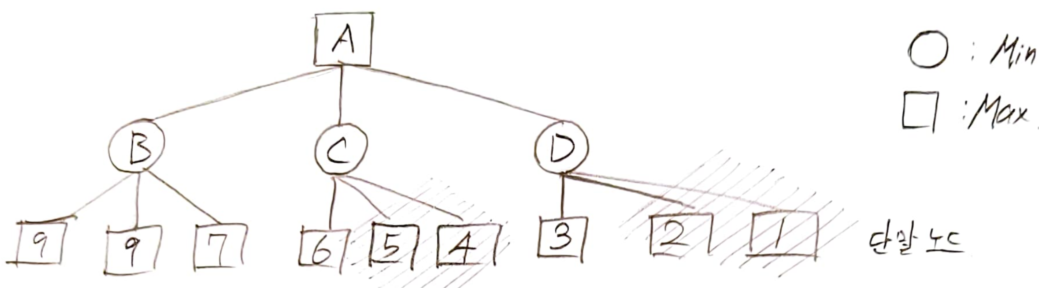
미니맥스 알고리즘의 최대 단점 : 시간 복잡도가 지수적으로 증가함. 경우의 수를 다 파악해야 하므로

시간 복잡도를 줄이기 위해 고민하다, "어떤 상태는 확장을 해도 목표 상태에 도달할 수 없다" 라는 것이 있다는 생각을 할 수 때, 이 "가능성 없는 것" 들을 판단만 하면 굳이 그 쪽으로 확장할 필요 X (확장 제외)

Like 트리에서 뻗어나갈 때 필요없는 가지는 쳐내는 것처럼... → 최적화에 높은 영향을 줌

알파베타 가지치기 : 위 내용과 같이 고안됨 ; 가지치기에 적용되는 파라미터 값으로 알파, 베타 사용함

↓ 평가값을 결정하는 데 드는 영향



1. 평가값을 계산한 상태 (단말 노드), Max는 A에서 BCD 확장, BCD는 각각 3개의 단말노드 확장

2. B : 3개 中 7 선택
   C : 3개 中 4 선택 → Max : 7, 4, 1 中 7 선택
   D : 3개 中 1 선택

   ⟹ 기존의 미니맥스 알고리즘 9가지 "전부 확인" 후 결정. (경우의 수 기하급수적 증가 가능)

3. 알파베타 가지치기라면?
   DFS 시에 맨 왼쪽부터 확장 후 탐색 진행. 처음 본 것이 9 → 9 잠정적 선택. (초기는 inf : infinity)
   두번째도 9, 세번째는 7 → 9가 7로 update.
   즉, Min은 Min이 선택 가능한 경우를 봤을 때 작은 값을 선택해야 하므로 작은 값 나올 때마다 update.
   B : inf → 9 → 9 → ⑦ B는 전부 탐색 완료.
   C : inf → 6
   근데 6을 생각해보니 앞의 7과 비교했을 때 작음. ⟹ B보다 어떻게든 작게 되므로 그 위 Max가 선택할 확률 X.
   6 이후 노드들이 6보다 클 수도, 작을 수도 있다. 일단 6보다 작은 수가 올라갈 테니 위로 가도 선택 가능성 X.
   그러므로 6 이후를 탐색하지 않음. (더이상의 update X)
   D : inf → 3. 과정 동일요

   ∴ Pruning : 선택할 확률이 없는 노드들을 탐색에서 제외시킴.
   Max가 선택할 수 있는 최대치와 Min이 선택할 수 있는 최소치를 가지고 그 외의 것들은 가지치기를 함요
   (α값) (β값)

   9 → 5개로 탐색 노드 축소 (기존의 55%) ; 시간 축소요

알파베타 가지치기 : 탐색을 할 때 알파값과 베타값을 가식 노드로 관리

→ 가식 노드 : α, β값 비교하며 불필요한 탐색을 줄이

- Max : α값만을 업데이트 (자신이 선택 가능한 최대 값) → 알파 컷
- Min : β값만을 업데이트 (자신이 선택 가능한 최소 값) → 베타 컷

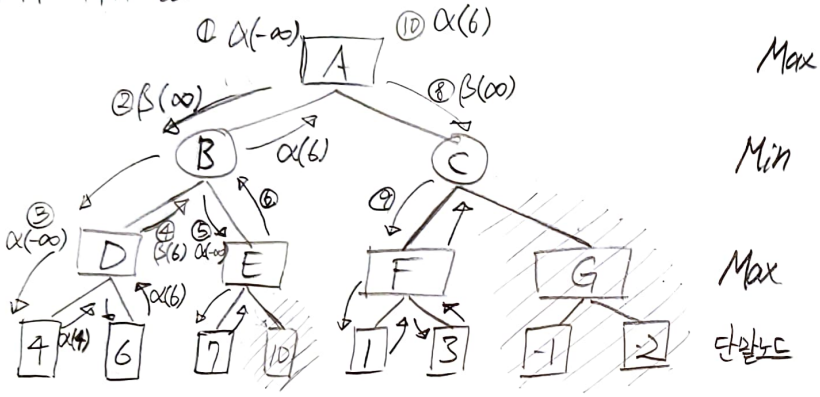
알고리즘 : 미니맥스 알고리즘의 최적화 버전 ; 게임 트리 상의 필요 없는 가지들을 잘라낼 수 있으므로 빠른 탐색 가능.

→ 사전 분석된 후행 노드들의 방문 순서에 크게 의존. (알 수 없이, 무작위로 탐색한다면 달라질 수)

- 미니맥스 : $O(b^m)$
- 알파베타 가지치기 : $O(b^{\frac{m}{2}})$

알고리즘 pseudocode : Inf:Gae_W3_03.txt 참고

알파베타 가지치기 실습



① α 최설정 ② B: β 최설정 ③ D 호출 : 4 노드 호출, 4 리턴 ; α값 4 update

6 노드 호출, 6 리턴 ; α값 6 update

⇒ α(6) < β(∞) , β컷 X. 이후 6 리턴 (α)

④ B 복귀, 리턴값 6으로 β교체 ⑤ E 탐색 ; 최초 α(-∞), β(6)

7 노드 호출, 7 리턴 ; α값 7 update

⇒ α(7) > β(6) ; break → β컷 발생 이하 노드 탐색 X 7 리턴

⑥ B 복귀, β(6) vs 7 → β(6), ∴ 6 리턴.

⑦ A 복귀 : α(6), β(∞) 상태 ⑧ C 탐색 : 최초 α(6), β(∞)

⑨ F 탐색 : 최초 α(6), β(∞) 1 노드 호출, 1 리턴 ; ⇒ 3 리턴 후 value(3) vs β(∞) → β(3) update

3 노드 호출, 3 리턴 ; value

α(6) > β(3), ∴ α컷 발생 → G에 3 리턴.

⑩ 최종 : α(6), ∴ A는 B노드 선택 경로를 갈 것이다. 노드탐색 10/14 만 진행됨. 약 30% 절약.

⇒ 가지치기 난이도↑ parameters 개념을 해가면서 체크해야 할듯 α컷 β컷 상관없이 α≥β면 Cut임 중요!!

어느 노드에서 알파나 베타에 따라 컷 여부 달라짐

05. 불완전한 결정

미니맥스 알고리즘의 한계 : 탐색 공간 전체를 탐색하는 것을 가정 ; 전체 탐색 불가 → 실제로는 적당한 시간 내 다음 수 결정

대안 : 탐색을 끝내야 하는 시간 도달 시 탐색 중단 & 탐색 중인 상태에 대해 휴리스틱 평가 함수 적용.

→ 비단말 노드이지만 단말 노드에 도달한 것으로 취급.

Why? 시간 안에 해결해야 하니까.

인공지능 → 사람 흉내내며 컴퓨터 특징 극대화하는 가치. 극대화하려는 알고리즘이 시간적 제약에 빠지다 보니 인공지능에게 100% 정확도보단 허용 가능한 오차를 가진 인공지능을 많이 사용함.

평가함수의 설정 : 비단말노드에서 확률을 보장할 수 있는 단계가 필요함. 보장할 수 있는 값을 잘 높여주려면 평가함수를 잘 만들어야

→ 인간이 만든 축 넣다보니 경험적 방법에 주로 의존함.

완벽한 평가 기능 실제는 불가능하나 평가 기능 향상 시 게임 플레이 향상.

평가 함수 설계 원칙 : 1. 승리 > 무승부 > 패배 상태 순으로 추정치가 높아야 한다.

→ 우/패가 많아지면 갈 수 있는 길이 별로 없음.

2. 추정치 계산에 너무 많은 시간이 걸리면 안된다. "간단해야 함"

→ 알고리즘 성능 떨어짐.

3. 실제 승리 확률과 유사한 추정치를 반환해야 한다

→ 잘이 안될 수 있으니까. 엉뚱한 루트 탈 수 있음.