Machine Learning in Practice

#7-2: Search Heuristics for Combinatorial Games

Summer 2019

1 Minimax Search

다음과 같이 게임의 전체 상태를 탐색하여 게임의 승자와 필승 전략을 계산할 수 있다.

```
in_W_A(c)
    if c ∈ F_B
        return F
    for each c' ∈ m_A(c)
        if in_W_B(c') is F
            return T
    return F
```

```
in_W_B(c)
    if c ∈ F_A
        return F
    for each c' ∈ m_B(c)
        if in_W_A(c') is F
            return T
    return F
```

바둑과 같은 실제 조합적 게임의 전체 상태의 크기는 매우 커서 위와 같이 전수 탐색(exhaustive search)을 적용하는 것이 가능하지 않고 다음과 같이 탐색 공간을 크게 줄여야 한다.

- 게임 탐색 트리의 너비(width)를 줄이기: $m_i(c)$에 포함되는 다음 상태를 전부 고려하지 않고 유리한 것으로 판단되는 상태만 고려해서 탐색하기
- 게임 탐색 트리의 깊이(depth)를 줄이기: 게임이 종료될 때까지 탐색하지 않고 중간에 적절히 판세를 판단하여 리턴하기

게임 탐색 트리의 너비를 줄이는 방식의 대표적인 예로는 3절에 다룰 MCTS(Monte-Carlo Tree Search) 기법이 있고, 깊이를 줄이는 방식으로는 미니맥스(minimax) 탐색과 이의 효율적 형태인 α-β 탐색 방식이 있다.

```
max_value_A(c, depth)
    if depth ≥ threshold
        return v(c) // value of c
    max_v := −∞
    for each c' ∈ m_A(c)
        v' := min_value_B(c', depth+1)
        max_v := max{max_v, v'}
    return max_v
```

```
min_value_B(c, depth)
    if depth ≥ threshold
        return v(c) // value of c
    min_v := ∞
    for each c' ∈ m_B(c)
        v' := max_value_A(c', depth+1)
        min_v := min{min_v, v'}
    return min_v
```

미니맥스 탐색 알고리즘은 위와 같이 탐색 깊이가 최대 허용치를 넘게 되면 탐색을 종료하면서 그 상태의 가치(value)를 추정한 값을 리턴한다. 탐색 깊이 허용치를 무한히 늘려주고 상태의 가치를 이기면 $+1$, 지면 -1로 설정해주면 전수 탐색과 일치하게 된다.

미니맥스 기법은 체스, 장기와 같이 각 상태의 가치를 적절한 수준에서 추정할 수 있는 경우 효율적인데, 바둑과 같이 게임의 종료 지점에 근접해야 가치를 판단할 수 있는 경우에는 적용하기 힘들다.

2 α-β Pruning

미니맥스 탐색은 게임 트리를 깊이우선탐색(depth-first search) 순서로 재귀적으로 탐색하게 되는데 탐색의 각 시점까지 계산한 가치(value) 정보를 이용하면 이후의 불필요한 탐색을 줄일 수 있다. α-β 탐색은 미니맥스 탐색과 동일한 결과를 계산하는데 변수 α와 β에 핵심적인 가치 정보를 보관하고 이 값들을 기반으로 불필요한 탐색 경로를 제외시킨다.

```
max_value_A(c, α, β, depth)
   if depth ≥ threshold
      return v(c) // value of c
   for each c' ∈ m_A(c)
      v' := min_value_B(c', α, β, depth+1)
      α := max{α, v'}
      if β ≤ α
         break  // β cut-off
   return α
```

```
min_value_B(c, α, β, depth)
   if depth ≥ threshold
      return v(c) // value of c
   for each c' ∈ m_B(c)
      v' := max_value_A(c', α, β, depth+1)
      β := min{β, v'}
      if β ≤ α
         break  // α cut-off
   return β
```

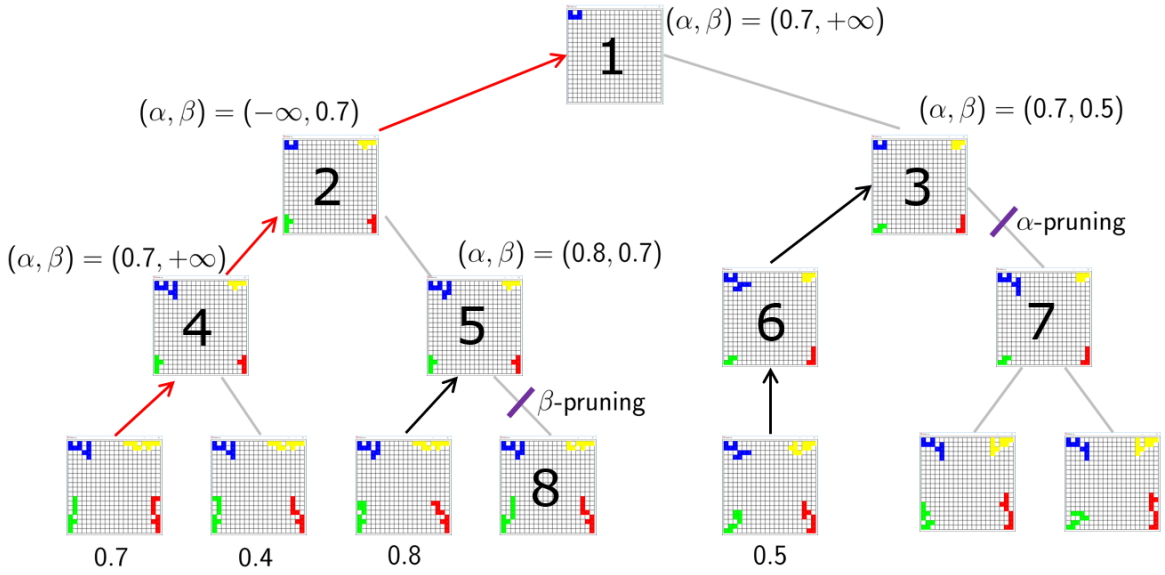
```
// initial call (from player A)
α := −∞
β := ∞
max_value_A(c_0, α, β, 0)
```

α-β 탐색에서의 α와 β는 각각 미니맥스 탐색에서의 *max_v*와 *min_v*에 대응되는데 다음과 같이 추가의 역할을 하게 한다.

- α: 지금까지 (DFS로) 탐색한 상태들 중 플레이어 A 입장에서의 최선의 가치(최대 value)로 A에게 유리한 상황의 최소값
- β: 지금까지 (DFS로) 탐색한 상태들 중 플레이어 B 입장에서의 최선의 가치(최소 value)로 A에게 불리한 상황의 최대값

변수 α와 β를 이용하여 다음과 같이 불필요한 탐색 경로를 가지치기(pruning)한다.

- α cut-off: 탐색 트리 상에서의 A가 시작할 어떤 서브트리(subtree)가 이전에 구해놓은 A 입장에서의 최선의 가치(α)와 비교했을 때 A에게 더 불리한 경우에 적용 (A가 이 서브트리를 택할 이유가 없으므로 B도 이 경로를 따라 탐색할 필요 없음)
- β cut-off: 탐색 트리 상에서의 B가 시작할 어떤 서브트리가 이전에 구해놓은 B 입장에서의 최선의 가치(β)와 비교했을 때 A에게 더 유리한 경우에 적용 (B가 이 서브트리를 택할 이유가 없으므로 A도 이 경로를 따라 탐색할 필요 없음)



루트 노드에서 탐색이 시작될 때 $(\alpha, \beta) = (-\infty, \infty)$로 초기화 된다. 탐색이 트리의 아래 방향으로 내려가면서 상위 노드의 α, β 값이 자식 노드로 전달되며, 서브트리의 탐색이 종료되어 상위 노드로 리턴할 때 변경된 α, β 값이 반영된다. 위 그림의 탐색 트리에서 노드들은 DFS 순서에 의해 $1, 2, 4, 2, 5, 2, 1, 3, 6, 3, 7, 3, 1$ 순서로 방문되는데, 노드 1에서 시작하는 플레이어는 A라고 가정하고 각 절차마다 (α, β)의 변화와 가지치기가 일어나는 상황을 살펴보자. (이때, 루트에서 짝수 거리에 있는 노드들(루트 토함)의 경우 α 값이 노드에 대응되는 상태의 가치가 되며, 홀수 거리에 있는 노드들의 경우 β 값이 가치가 됨을 염두에 두자.)

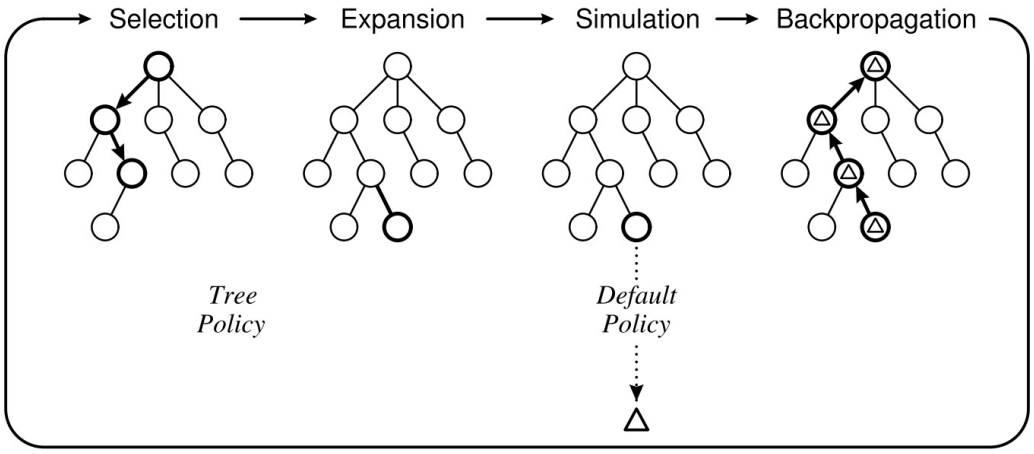
- $1 \to 2 \to 4$: $(\alpha, \beta) = (-\infty, \infty)$가 그대로 전달된다.
- 4: 플레이어 A는 가치가 0.7인 왼쪽 자식 노드의 상태를 선택하면서 $\alpha = 0.7$을 리턴받아서 $(\alpha, \beta) = (0.7, \infty)$가 된다.
- $4 \to 2$: 노드 4에서의 $\alpha = 0.7$이 노드 2의 β값으로 들어가면서 $(\alpha, \beta) = (-\infty, 0, 7)$이 된다.
- $2 \to 5$: $(\alpha, \beta) = (-\infty, 0, 7)$가 그대로 전달된다.
- 5: 플레이어 A는 왼쪽 자식 노드로부터 $\alpha = 0.8$을 리턴 받아 $(\alpha, \beta) = (0.8, 0.7)$이 된다.
- $5 \to 8$: 오른쪽 자식 노드의 가치가 0.8보다 작으면 노드 5의 가치는 0.8로 유지되고 0.8보다 크면 그 값으로 노드 5의 가치가 변경된다. 그런데 어느 경우에도 노드 2에서 노드 5의 가치는 노드 4의 가치와 min으로 병합되므로 노드 2의 플레이어인 B에 의해 선택되어질 이유가 없다. 따라서 노드 8은 방문할 필요가 없어진다. (다르게 말하면 $\alpha = 0.8 \geq 0.7 = \beta$므로 β cut-off를 적용한다.)
- $5 \to 2$: $\alpha = 0.8$이 노드 2로부터 리턴되어 노드 1에 저장된 $\beta = 0.7$과 비교되는데 min으로 병합되므로 $\beta = 0.7$이 변경되지 않고 $(\alpha, \beta) = (-\infty, 0, 7)$가 그대로 유지된다.

- $2 \rightarrow 1$: 노드 2에서의 $\beta = 0.7$이 노드 1의 α값으로 들어가면서 $(\alpha, \beta) = (0.7, \infty)$가 된다.
- $1 \rightarrow 3$: $(\alpha, \beta) = (0.7, \infty)$가 그대로 전달된다. 나중에 노드 3에서 리턴할 가치(β)가 노드 2의 가치인 0.7보다 크지 않으면 노드 1에서 플레이어 A가 선택받지 않음에 주목한다.
- $3 \rightarrow 6 \rightarrow 3$: 노드 6으로 부터 $\alpha = 0.5$를 리턴받아 노드 3의 $\beta = 0.5$가 되어 $(\alpha, \beta) = (0.7, 0.5)$가 된다.
- $3 \rightarrow 7$: 오른쪽 자식 노드인 7의 가치가 0.5보다 크면 노드 3의 가치는 0.5로 유지되고 0.5보다 작으면 그 값으로 노드 3의 가치가 변경된다. 그런데 어느 경우에도 노드 1에서 노드 3의 가치는 노드 2의 가치와 max로 병합되므로 노드 1의 플레이어인 A에 의해 선택되어질 이유가 없다. 따라서 노드 7은 방문할 필요가 없어진다. (다르게 말하면 $\beta = 0.5 \leq 0.7 = \alpha$므로 α cut-off를 적용한다.)

3 Monte-Carlo Tree Search

바둑의 경우 체스, 장기 등의 게임과 달리 게임의 종료 지점에 근접하기 전에는 각 상태의 가치를 추정하는 것이 거의 불가능하여 α-β 탐색을 적용하기 힘들다. 즉, 이와 같은 경우를 위해서는 탐색 트리의 깊이를 줄이지는 않고 너비를 줄이는 방식을 고려해야 하는데, MCTS(Monte-Carlo Tree Search) 기법이 이에 해당된다.

MCTS의 이름에 Monte-Carlo가 들어가는 것을 보면 알 수 있듯, 이 방법은 게임 트리의 탐색 경로를 랜덤하게 선택하는 것을 충분히 많은 횟수만큼 반복하여 통계적으로 최적에 가까운 수를 찾는 방식이다. 고수준에서 MCTS를 살펴보면, 다음 그림과 같은 4가지 절차를 계속 반복하는 과정이라 할 수 있다.



각 절차에 대한 설명은 다음과 같다. 여기에서 핵심은 tree policy와 default policy이다.

1. Selection: 루트 노드에서부터 tree policy(child selection policy)를 재귀적으로 적용해서 말단 노드 L까지 도달한 후 L을 선택한다.
2. Expansion: 도달한 말단 노드 L이 게임의 최종 상태가 아니라면 tree policy(leaf create policy)를 적용하여 새로운 자식 노드 하나 또는 여러 개를 더 만들어서 트리를 확장한다.
3. Simulation: 새 노드에서 default policy에 따른 결과를 랜덤 시뮬레이션을 통해 계산한다.
4. Backpropagation: 시뮬레이션 결과를 사용해 selection 단계에서 사용하는 정보들을 수정한다.

이때 tree policy와 default policy는 다음과 같다.

- Tree policy: 이미 존재하는 탐색 트리에서 말단 노드를 선택하거나 생성하는 policy
  - 바둑의 경우에는 특정 시점에서 가능한 모든 수 중에서 가장 승률이 높은 수를 예측하는 policy라고 생각하면 된다.
- Default policy: 주어진 (최종 노드가 아닌) 노드에서의 (얼마나 좋은 상태인지를 측정하는) 가치를 추정을 하는 policy
  - 바둑의 경우에는 현재 상황에서 얼마나 승리할 수 있을지를 측정하는 policy라고 생각하면 된다.

Backpropagation 절차 자체는 둘 중 어떤 policy도 사용하지 않지만, 대신 backpropagation을 통해 각 policy들의 파라미터들이 수정된다. 이 4개의 절차가 하나의 단위로, MCTS는 시간이 허락하는 한도 내에서 이 과정을 계속 반복하고, 그 중에서 가장 좋은 결과를 자신의 다음 action으로 삼는다. 다음은 가장 좋은 노드를 고르는 기준의 4가지 예시이다.

1. Max child: 가장 높은 보상(reward) 값을 가지고 있는 노드를 고른다.
2. Robust child: 루트 노드에서부터 가장 많이 방문된 노드를 고른다.
3. Max-Robust child: 1, 2를 동시에 만족하는 노드를 고르며, 그런 노드가 없다면 계속 반복해서 그런 노드를 찾아낸다.
4. Secure node: 가장 lower confidence bound를 최대화하는 노드를 고른다.

MCTS의 일반적인 알고리즘을 정리하면 다음과 같이 적을 수 있다. (1) tree policy, (2) default policy, (3) best child selection 이 세가지를 어떻게 정하느냐에 따라서 구체적인 알고리즘이 결정된다.

function MCTSSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD(v_0))

4 MCTS with UCT

가장 널리 사용되는 UCT(Upper Confidence Bounds) 방식의 MCTS에서는 exploitation과 exploration을 다음 식을 이용해 조정한다.

$$\underset{v' \in child(v)}{\operatorname{argmax}} \left(\underbrace{\frac{Q(v')}{N(v')}}_{\text{exploitation}} + \underbrace{c\sqrt{\frac{2\ln N(v)}{N(v')}}}_{\text{exploration}} \right)$$

- $N(v')$: 노드 v'를 지나가는 실험 횟수 ($N(v) = \sum_{v' \in child(v)} N(v')$)
- $Q(v')$: 노드 v'를 지나갔을 때 이긴 실험 횟수

로 $N(\cdot), Q(\cdot)$를 정의하면

- $\frac{Q(v')}{N(v')}$: 노드 v'를 택했을 때 이길 확률
- $\sqrt{\frac{2\ln N(v)}{N(v')}}$: 노드 v'쪽으로 실험이 적게 이루어질수록 커짐

로 exploitation과 exploration을 조정하는데 이길 확률이 높은 쪽으로 더 자주 실험하되(exploitation) 실험이 충분히 되지 않은 쪽으로도 실험을 유도(exploration)하는 것이다. 이 방식에서 횟수를 무한히 늘리면 최적에 수렴함은 증명되어 있고, 전체 알고리즘은 아래와 같다.

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

알고리즘을 보면 알 수 있듯, 강화학습스러운 방식을 취하기 때문에

- Aheuristic: 뭔가 이유가 있고 합리적인 의사결정을 할 수 있으며
- Asymmetric: 아래 그림처럼 tree를 대칭적이지 않게, 더 중요한 부분만 집중적으로 탐색할 수 있다.