National University of Singapore
School of Computing
CS3243 Introduction to AI

Tutorial 6: Adversarial Search (Solutions)

---

1. Consider the minimax tree given in Figure 1. In the figure, black nodes are controlled by the MAX player, and white nodes are controlled by the MIN player. Payments (terminal nodes) are squares; the number within denotes the amount that the MIN player pays to the MAX player (an amount of 0 means that MIN pays nothing to MAX). Naturally, MAX wants to maximize the amount they receive, and MIN wants to minimize the amount they pay.

   Suppose that we use the α–β `PRUNING` algorithm, given in Figure 6.7 of AIMA 4th edition (reproduced in Figure 2).

   (a) **Assume that we iterate over nodes from right to left**; mark with an 'X' all edges that are pruned by α-β pruning, if any.

   (b) **Assume that we iterate over nodes from left to right**; mark with an 'X' all edges that are pruned by α-β pruning, if any.

   (c) Determine if the effectiveness of pruning depends on iteration order (i.e., provide a **yes or no** answer. (There is no need to elaborate on this answer for your assignment submission However, you should consider why iteration order is important for discussion during the tutorial.)

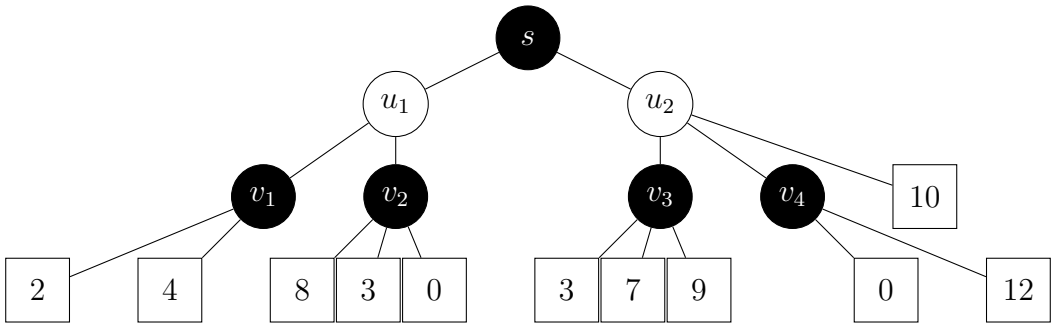


Figure 1: A minimax search tree.

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Figure 2: α–β `PRUNING` algorithm.

Solution: We refer to the players' actions as R (right) M (middle) and L (left). We let $\alpha(x)$ be the least payoff that MAX can guarantee at node x, and $\beta(x)$ be the maximal payoff MIN has to pay at node x.

Let us consider a run of the α–β `PRUNING` algorithm here. First, the MAX player will explore R, followed by the MIN player exploring R. At this point, $\beta(u_2) = 10$.

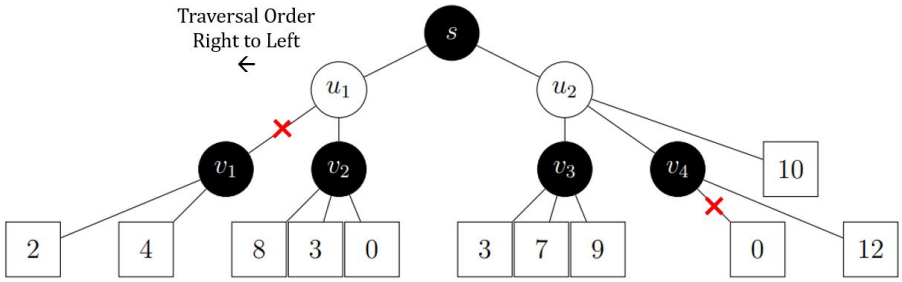
Next, MIN plays M and explores v_4; MAX explores R and observes a value of 12 (so $\alpha(v_4) = 12$); since this value is greater than $\beta(u_2)$ we know that choosing M at u_2 yields a lower value than playing R, the edge between v_4 and 0 is pruned. Next, MIN explores L (the node v_3), seeing the values 9, 7 and 3 in this order; this yields $\alpha(v_3) = 9$), and we update $\beta(u_2) = 9$, and subsequently $\alpha(s) = 9$ (in other words, if MAX chooses R at s they can get a payoff of at least 9).

Next, MAX explores L at s, MIN explores R at u_1, and MAX explores R, followed by M and L, at v_2. This yields $\alpha(v_2) = 8$, which means that $\beta(u_1) = 8$ which is strictly smaller

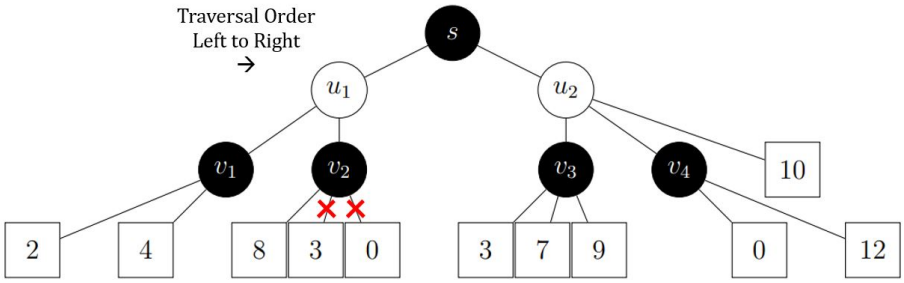
than $\alpha(s)$. What this immediately implies is that the action L is strictly worse for MAX than the action R, and there is no point in continuing to explore its subtree: if MAX chooses L then MIN can choose R and obtain a better outcome (for MIN). At this point, there is no more need to explore the action L at u_1 (and its subtree) and the algorithm stops: the optimal action sequence is R at s, L at u_2 and R at v_3, with MIN paying 9 to MAX.

Next, we assume that agents explore actions from left to right. The MAX player first explores (s, u_1), followed by the MIN player exploring (u_1, v_1), and the MAX player checking $(v_1, 2)$; then we check $(v_1, 4)$ and $\alpha(v_1) = 4$ which updates $\beta(u_1) = 4$ as well (in the worst case the MIN player loses 2 here by playing L). Then we check (u_1, v_2) and $(v_2, 8)$. At this point we stop - $\alpha(v_2) = 8$ but $\beta(u_1) = 4$! We prune both $(v_2, 3)$ and $(v_2, 0)$. Onto checking (s, u_2): we next check (u_2, v_3) and $(v_3, 3)$, $(v_3, 7)$ and $(v_3, 9)$, yielding $\alpha(v_3) = \beta(u_2) = 9$. Choosing (u_2, v_4) next, we check both $(v_4, 0)$ and $(v_4, 12)$ to get $\alpha(v_4) = 12$ and $\beta(u_2)$ remains at 9. Finally, we check $(u_2, 10)$ and nothing changes.

This is summarised by the following search tree.



Going from left to right instead, the result is summarised as follows.



As we can clearly see, order of traversal makes a difference.

2. Consider the following game: we have an attacker looking at three targets: t_1, t_2 and t_3. A defender must choose which of the two targets it will guard; however, the attacker has an advantage: it can observe what the defender is doing before it chooses its move. If an attacker successfully attacks it receives a payoff of 1 and the defender gets a payoff of -1.

   Model this problem as a minimax search problem. Draw out the search tree. What is the defender's payoff in this game?

   Solution: The search tree is given in Figure 3. Note that each defender action leads to a node it is *not* defending, denoted $\neg t_i$.

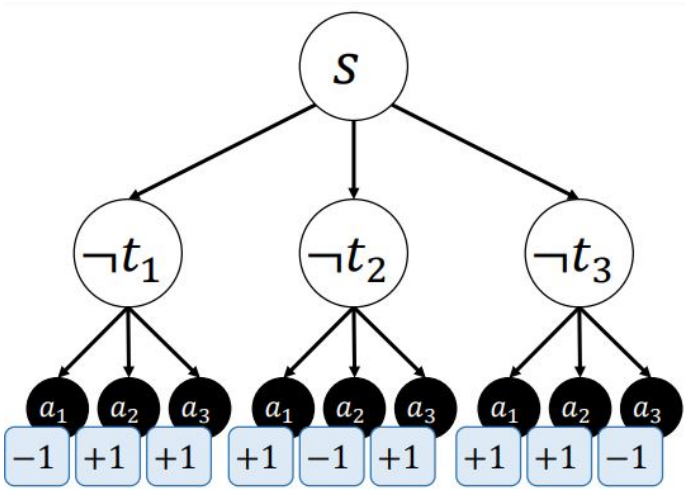


Figure 3: The game tree.

   Note that the attacker can choose an action that guarantees it a payoff of 1 no matter what the defender does. However, when the defender randomizes the attacker's benefit becomes significantly smaller. By choosing the target not to defend with probability $\frac{1}{3}$ its expected loss becomes $\frac{1}{3}$. Note that when the number of targets is n, this probability is $\frac{1}{n}$.

3. With the `MINIMAX` algorithm, we know that the value v, computed at the root (i.e, the utility for the MAX player), is a worst-case value. This means that if the opponent MIN does not act optimally, the actual outcome v' for MAX can only be better, and never worse than v. That said, the `MINIMAX` algorithm may not select the optimal move given sub-optimal play from the MIN player.

   Construct an example where, should the MIN player play sub-optimally, the `MINIMAX` algorithm makes a sub-optimal move.

   Solution: An example is depicted in Figure 4. Based on the `MINIMAX` algorithm, MAX plays $s \mapsto u_2$. However, assuming MIN plays sub-optimally, $s \mapsto u1$ should instead be picked for the +10 payout.

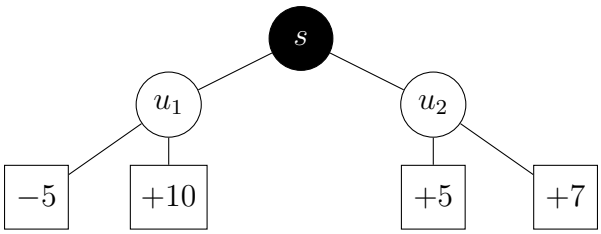


Figure 4: An example game tree when MAX may exploit sub-optimal play.