# 19 Optimal behavior: Game theory

Adversarial state dynamics
– have to account for *worst* case

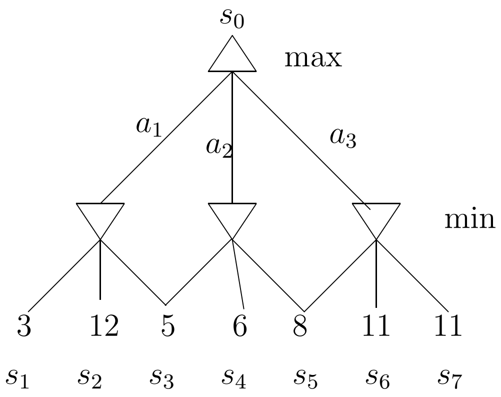
Compute policy $\pi : S \rightarrow A$ that maximizes *minimum* reward

Let $S'(a, s) = \{$ set of possible states $s'$ reachable by doing $a$ in $s$ $\}$

(Assume you can identify current state)

## 19.1 Single step case: Maxi-min reward

E.g.



Sets of possible next states

$$\begin{aligned} S'(a_1, s_0) &= \{s_1, s_2, s_3\} \\ S'(a_2, s_0) &= \{s_3, s_4, s_5\} \\ S'(a_3, s_0) &= \{s_5, s_6, s_7\} \end{aligned}$$

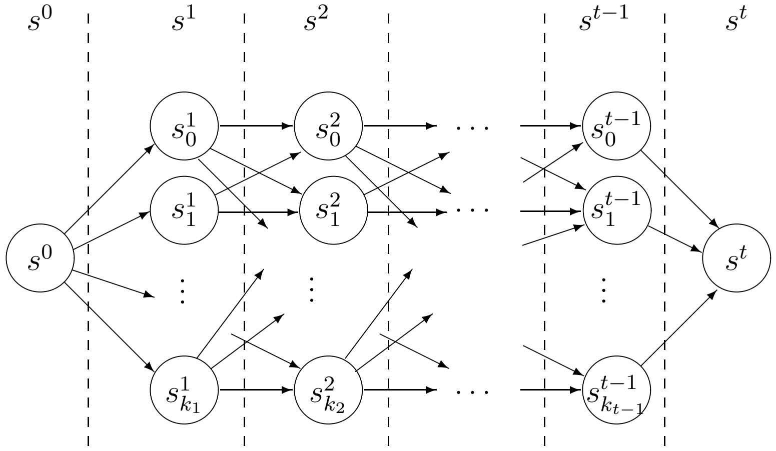
Solve

$$\begin{aligned} a^* &= \arg\max_a \min_{s' \in S'(a, s_0)} R(s') \\ &= a_3 \end{aligned}$$

Obtain a reward of 8

## 19.2 Sequential acyclic case

Assume leveled acyclic model (as in acyclic decision theory case)



Assume

- Start at state $s^0$ and finish at state $s^t$ after $t$ actions
- Model is acyclic:
- after executing action in $s^0$ we go to one of the states

  $$s^1_0, s^1_1, \ldots, s^1_{k_1}$$

  and after executing the second action, we go to one of the states

  $$s^2_0, s^2_1, \ldots, s^2_{k_2}$$

  and so on, until after the $t$th action, we arrive in state $s^t$

Given

- State dynamics $S'(a, s)$
- Reward function $R(s)$

Compute

- Optimal policy $\pi^* : S \to A$
  maximizes minimum total future reward for each state

### Utility function

$$\begin{aligned} U(s,\pi) &= \text{minimum future reward obtained by running } \pi \text{ from } s \\ &= R(s) + \min_{s' \in S'(\pi(s),s)} U(s',\pi) \end{aligned}$$

Compute $\pi^*$ that maximizes $U(s,\pi^*)$ for all $s$

### Efficient algorithm: Dynamic programming

Solve for $U(s,\pi)$ in last states first, and then recursively back up

$$\begin{aligned} \pi^*(s^i) &= \arg\max_a R(s^i) + \min_{s^{i+1} \in S'(a,s^i)} U(s^{i+1},\pi^*) \\ &= \arg\max_a \min_{s^{i+1} \in S'(a,s^i)} U(s^{i+1},\pi^*) \\ U(s^i,\pi^*) &= R(s^i) + \min_{s^{i+1} \in S'(a,s^i)} U(s^{i+1},\pi^*) \end{aligned}$$

where $U(s^{i+1},\pi^*)$ is already computed

## 19.3 Special case: Two-person Zero-sum game

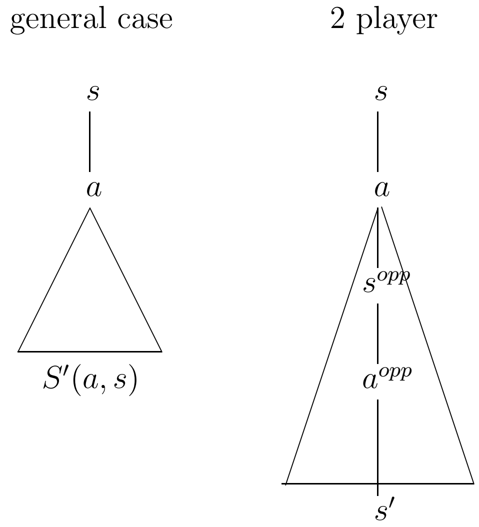
Assume

- Acyclic state dynamics
- 2 players
  - MAX player $\triangle$ tries to *maximize* reward
  - MIN player $\triangledown$ tries to *minimize* reward
- $R(s) = 0$ except at leaf states

Then can dramatically speed up dynamic programming by $\alpha$-$\beta$ *pruning*

**Note: slight augmentation in dynamics**

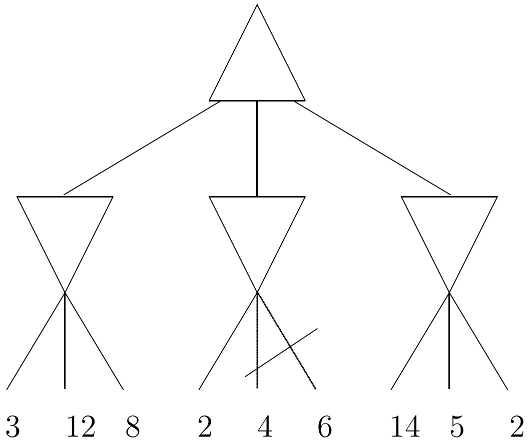
Now explicitly model opponent's moves



### $\alpha$-$\beta$ pruning

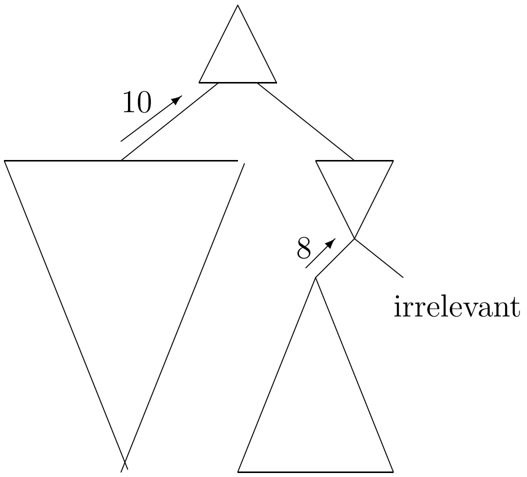
Not every path has to be explored

E.g.



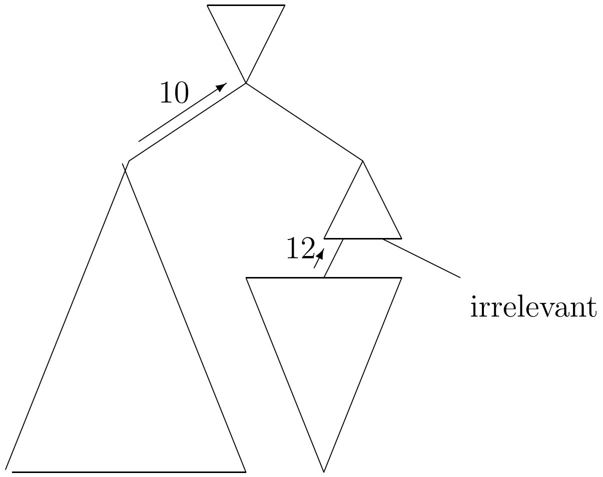
Assume the nodes are explored left to right in a depth first fashion. Once the children of the left MIN node are explored, the left MIN node will choose reward 3. The MAX node at the top will then know that it can obtain reward at least 3. The second MIN node (in the middle) will then start to explore its children. Once it sees that its first child has value 2, it knows that whatever the other children return, it can only return a reward that is 2 or smaller. But this means the rest of the children of the middle MIN node are irrelevant, because they cannot cause this node to obtain a larger value than 2. So the top MAX node will ignore the rest of the middle subtree because the MAX node can already achieve reward 3 elsewhere. Therefore, the leaves with rewards 4 and 6 are irrelevant and we do not need to check them.

**$\alpha$-cutoff**



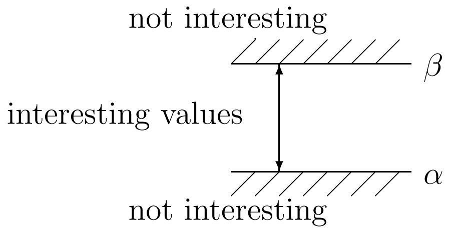
$$\begin{aligned}\alpha &= \text{cutoff value for MIN node (if current best value} \leq \alpha\text{, stop)}\\ &= \text{current highest value of MAX ancestor}\\ &= \text{lower bound on value MIN can hope to back up to root}\\ &= \text{10 in this example}\end{aligned}$$

**$\beta$-cutoff**



$$\begin{aligned}\beta &= \text{cutoff value for MAX node (if current best value} \geq \beta\text{, stop)}\\ &= \text{current lowest value of MIN ancestor}\\ &= \text{upper bound on value MAX can hope to back up to root}\\ &= \text{10 in this example}\end{aligned}$$

This method is called $\alpha$-$\beta$ pruning, and it is implemented by calculating the bounds $\alpha$ and $\beta$ of "interesting" values:



Algorithms 1 and 2 describe operations in max and min nodes for $\alpha$-$\beta$ pruning.

**Algorithm 1** $\alpha$-$\beta$ pruning algorithm: *alpha_beta_max*$(s, \alpha, \beta)$

```
Require: $s$ is a max-node situation, $\alpha$ and $\beta$ are boundaries
Ensure: the max value and an optimal action is returned, assuming given
    boundaries
if $s$ is a leaf node then
  return $(R(s)$, 'no action')
end if
opt_action ← 'not important'
for all possible actions $a$ do
  $(v, m) \leftarrow$ alpha_beta_min$(a(s), \alpha, \beta)$
  if $v > \alpha$ then
    $\alpha \leftarrow v$
    opt_action ← $a$
    if $\alpha \geq \beta$ then
      return $(\alpha$, opt_action$)$
    end if
  end if
end for
return $(\alpha$, opt_action$)$
```

If we know the minimal and maximal value of the reward, i.e., $\min_s R(s)$ and $\max_s R(s)$, then those are the initial values for $\alpha$ and $\beta$; otherwise, $\alpha = -\infty$ and $\beta = +\infty$ initially.

The figure 1 illustrates what values of $\alpha$ and $\beta$ are passed during the search, and which nodes are visited (circled ones), in $\alpha$-$\beta$ pruning (the initial values are $\alpha = -\infty$ and $\beta = +\infty$.

**Algorithm 2** $\alpha$-$\beta$ pruning algorithm: $\mathit{alpha\_beta\_min}(s, \alpha, \beta)$

```
Require: $s$ is a min-node situation, $\alpha$ and $\beta$ are boundaries
Ensure: the min value and an optimal action is returned, assuming given boundaries
if $s$ is a leaf node then
    return $(R(s),$ 'no action'$)$
end if
$opt\_action \leftarrow$ 'not important'
for all possible actions $a$ do
    $(v, m) \leftarrow alpha\_beta\_max(a(s), \alpha, \beta)$
    if $v < \beta$ then
        $\beta \leftarrow v$
        $opt\_action \leftarrow a$
        if $\alpha \geq \beta$ then
            return $(\beta, opt\_action)$
        end if
    end if
end for
return $(\beta, opt\_action)$
```

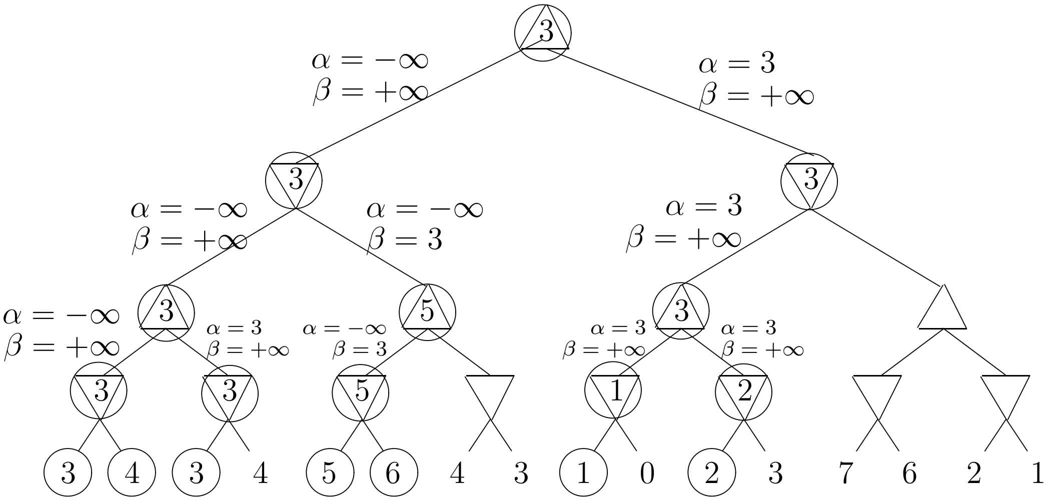


Figure 1: $\alpha$-$\beta$ pruning example

### Application to real games

A problem with almost all practical games is that the search tree is too large. If the game is cyclic, it is infinite. In any case, it is usually impossible in practice to search the whole tree. On improvement is to use *memoization,* i.e., keep a cache of calculated situations (positions). Additionally, memoization can prevent us from exploring in an infinite loop in cyclic state spaces.

The leaves of the search tree are usually to far away to be reached by a search algorithm, so we cannot usually back up exact values from the leaves. (Remember that the leaves are the terminal states at the very end of the game!) In practice, one almost always uses a heuristic approach: search to a bounded depth, evaluate a heuristic function at the states reached (which estimates the future min-max reward), treat this heuristic value as the terminal reward, and back up the results.

### Combined random-adversarial games

In some games, the sequence of states does not depend on the players' actions alone, but also on a random element, such as a dice roll or a card shuffle (e.g. Backgammon or Poker). In these cases, the game tree contains *chance* nodes in addition to MIN and MAX nodes. The value in chance nodes in calculated as the expected value of the child-nodes' values, using the probability distribution of the random event. In this case, we use the terms *expecti-mini-max algorithm, expecti-mini-max policy,* etc.

## 19.4 General cyclic case

Maximize discounted sum of minimum future rewards

### Value function

$$\begin{aligned} V_\pi(s) &= \text{minimum discounted reward obtained by } \pi \text{ starting in } s \\ &= R(s) + \gamma \min_{s' \in S'(\pi(s), s)} V_\pi(s') \end{aligned}$$

## Policy evaluation

- Initialize $V_\pi$ arbitrarily
- Iterate
$$V_\pi^{new}(s) = R(s) + \gamma \min_{s' \in S'(\pi(s),s)} V_\pi^{old}(s')$$
for each $s$
- Halt when $V_\pi^{new}$ and $V_\pi^{old}$ are sufficiently close

## Policy iteration

- Initialize $\pi$ arbitrarily and evaluate $V_\pi$
- Iterate
$$\begin{aligned} \pi^{new}(s) &= \arg\max_a \;\; R(s) + \gamma \min_{s' \in S'(a,s)} V_{\pi^{old}}(s') \\ &= \arg\max_a \;\; \min_{s' \in S'(a,s)} V_{\pi^{old}}(s') \end{aligned}$$
for each $s$
- Use policy evaluation to calculate $V_{\pi^{new}}$ for $\pi^{new}$
and repeat policy update
- Halt when $\pi^{new} = \pi^{old}$ (or more generally when $V_{\pi^{new}} = V_{\pi^{old}}$)

## Value iteration

- Initialize $V$ arbitrarily
- Iterate
$$V_{new}(s) \;\; = \;\; R(s) + \gamma \max_a \;\; \min_{s' \in S'(a,s)} V^{old}(s')$$
for each $s$
- Halt when $V^{new}$ and $V^{old}$ are sufficiently close

### Recovering greedy policy

Given a value function $V$, recover $\pi$ by

$$\begin{aligned} \pi(s) &= \arg\max_a \; R(s) + \gamma \min_{s' \in S'(a,s)} V(s') \\ &= \arg\max_a \; \min_{s' \in S'(a,s)} V(s') \end{aligned}$$

for each $s$

## Readings

Russell and Norvig: Sections 6.1, 6.2, 6.5