Games

Formalization and Algorithms

Introduction to Artificial Intelligence

Chandra Gummaluru
University of Toronto

Version W22.1

- The following is based on material developed by many individuals, including (but not limited to):
  - Sheila McIlraith
  - Bahar Aameri
  - Fahiem Bacchus
  - Sonya Allin

- Recall the components of a **game**:
  - N players, indexed 1 through N
  - a set of states, S, in which the game could be in
  - a set of terminal states, $T \subseteq S$ in which the game ends
  - a set of actions, $A_i(s)$, available to each player, i from each state $s \in S$:
    - each n-tuple, $(a_1, \ldots, a_N), a_i \in A_i(s)$ is a called a joint-action from s, and $A(s)$ denotes the set of all possible joint actions, i.e.,

$$A(s) = \prod_{j=1}^{N} A_j(s).$$

    - we use a_{-i} to denote the joint action between all players except i, and $A_{-i}(s)$ to denote the set of all possible joint actions between all players except i, i.e.,

$$A_{-i}(s) = \prod_{j \neq i} A_j(s)$$

    - we use $S(s, a)$ to denote the state resulting from applying $a \in A(s)$ to s
  - a reward function, $r_i : A \to \mathbb{R}$, for each player i, so that $r_i(a)$ is a measure of how useful an action a is to player i

- Given an $s_0 \in S$, each possible realization of the game is a sequence of actions, $\langle a^{(1)}, \ldots, a^{(n)} \rangle$, such that $a^{(k)} \in A(s_{k-1})$, $s_k = S\left(s_{k-1}, a^{(k)}\right)$ and $s_n \in T$.
- The dynamics of the game are determined by a policy function, p, where $p(a|s)$ is the probability that $a \in A(s)$ is played from state, s.
- To solve a game means to find p assuming the players are rational.
- We can define the the expected-reward (value) of taking the action, a_i, for player i, when the other players' joint action is a_{-i}, using the following recurrence:

$$v_i(a_i, a_{-i}) = \begin{cases} r_i(a_i, a_{-i}) & S(s, a_i, a_{-i}) \in T \\ r_i(a_i, a_{-i}) + \sum_{a' \in A(s')} p(a'|s') v_i(a') & \text{otherwise} \end{cases} \tag{ExpRwd}$$

where $a_i \in A_i(a)$, $a_{-i} \in A_{-i}(s)$, and $s' = S(s, a)$.

- If all players are rational, we would have

$$p(a|s) = \begin{cases} 1 & a = a^*(s) \\ 0 & \text{otherwise} \end{cases} \tag{OptPol}$$

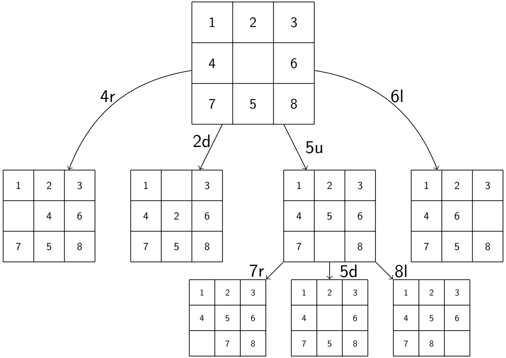
where $a^*(s)_i$ is player i's best-response to the joint action, $a^*(s)_{-i}$:

$$a^*(s)_i = \arg\max_{a_i \in A_i(s)} v_i\left(a_i, a^*(s)_{-i}\right). \tag{OptAct}$$

- It is difficult to find p in general, so we made two simplifying assumptions:
  - There is only one-player.
  - The set of terminal states, $T \subseteq S$, is replaced with a set of goal states, $G \subseteq S$
- We shall now slightly relax both of these assumptions:
  - there are now two-players
  - only one player, called the turn-taker, can act from any given state; rather than defining A_1 and A_2 separately, we simply define $A(s)$ to be the set of actions available to the turn-taker at s
  - the turn-taker switches after each action
  - we assume $r_1(a) = -r_2(a)$; rather than defining r_1 and r_2 separately, we let $r(a)$ to be the reward obtained by the turn-taker for taking action, a, and assume that r_1
  - $r(a) = 0$ for any $a \in A(s)$ such that $S(s, a) \notin T$

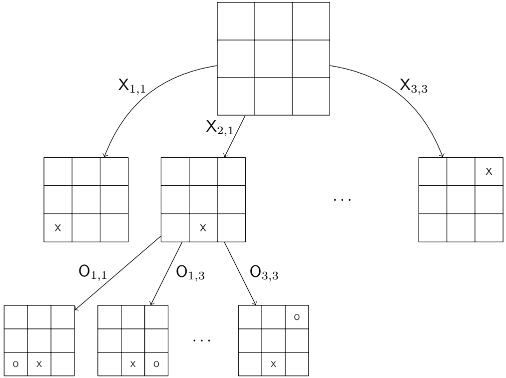
- We saw that a graph could be used to represent the structure of a search problem. However, the searching itself takes place on a tree of possible paths.

  Example: Search Tree for Slider Puzzle



- Similarly, we can create a tree of possible paths for our game. However in this case, each level corresponds to different player, namely, the turn-taker.

Example: Game Tree for Tic-Tac-Toe Puzzle

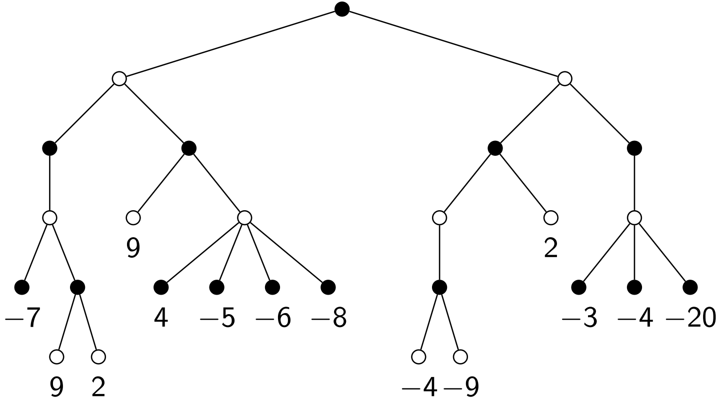


- Under the new assumptions, (ExpRwd) becomes:

$$v(a) = \begin{cases} r(a) & S(s,a) \in T \\ -\displaystyle\sum_{a' \in A(s')} p(a'|s)v(a') & \text{otherwise} \end{cases}$$

- The strategy of a player is defined by the family of distributions, $p(\cdot|s)$ for s in which they are the turn-taker.

- We will analyze various strategies on the tree below, where • and ∘ respectively denote states in which we are the turn-taker, or our adversary is:



- The rewards are expressed in the perspective of the player who last acted.

- If both players are perfect, i.e., p satisfies (OptPol), then (ExpRwd) reduces to:

$$v(a) = \begin{cases} r(a) & S(s,a) \in T \\ -\max\limits_{a' \in A(s')} v(a') & \text{otherwise} \end{cases} \qquad \text{(NegMax)}$$

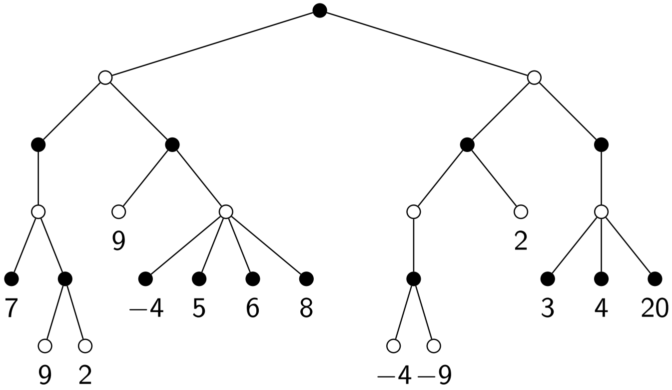
  where $s' = S(s, a)$.

- We can equivalently, express (NegMax) in terms of v_1 only, in which case, we have

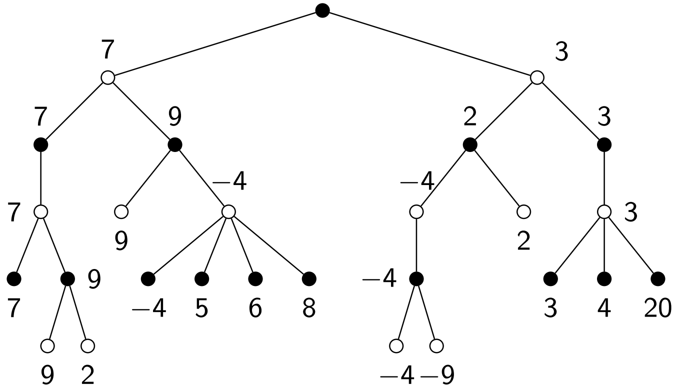
$$v_1(a) = \begin{cases} r_1(a) & S(s,a) \in T \\ \max\limits_{a' \in A(s')} v_1(a') & S(s,a) \notin T \text{ and we are the turn-taker} \\ \min\limits_{a' \in A(s')} v_1(a') & S(s,a) \notin T \text{ and we are not the turn-taker} \end{cases} \qquad \text{(MinMax)}$$

- We can define the value of states as follows: $v_1(S(s,a)) = v_1(a), \forall a \in A(s), s \in S$

- To use (MinMax), we need to first compute the rewards in our perspective.



- We can now compute the rewards of the actions as follows:



Game Strategies: MinMax Psuedocode

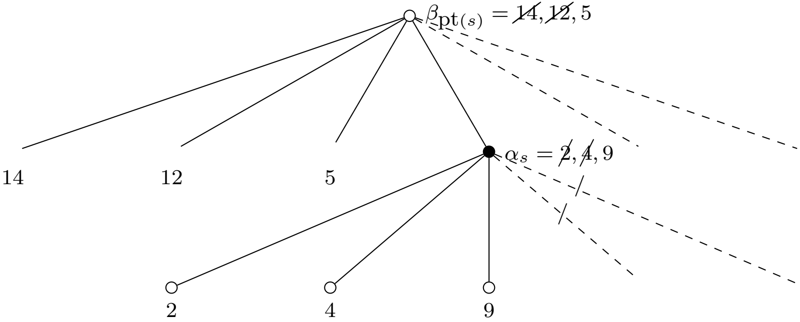
- For any non-terminal state, we recursively compute either the maximum or minimum of its successors' values.

procedure $\textsc{MinMaxSearch}(s, t)$
    if $s \in T$ **then** $\triangleright$ s is a terminal state
        return $r_1(s)$
    if $t = 1$ **then**
        $v_1(s) \leftarrow -\infty$
        for $a \in A(s)$ **do**
            $v_1(s) \leftarrow \max\{v_1(s), \textsc{MinMaxSearch}(S(s, a), 2)\}$
    if $t = 2$ **then**
        $v_1(s) \leftarrow \infty$
        for $a \in A(s)$ **do**
            $v_1(s) \leftarrow \min\{v_1(s), \textsc{MinMaxSearch}(S(s, a), 1)\}$
    return $v_1(s)$

- It turns out if we use a depth-first search, we do not necessarily need to explore the entire tree to compute the min-max utility of the root.

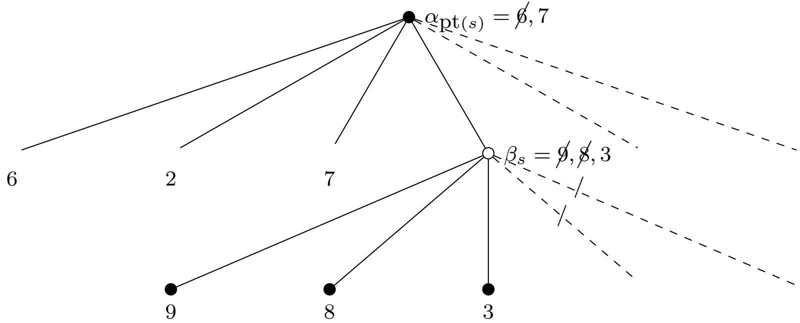
- Let s be some state and define two quantities:
  - α_s: the maximum utility guaranteed at s thus far.
  - β_s: the minimum utility guaranteed at s thus far.

- If we are the turn-taker at s:
  - α_s increases to the maximum value of s's successors as they explored.
  - $\beta_s = \beta_{\mathrm{pt}(s)}$ and will remain fixed.



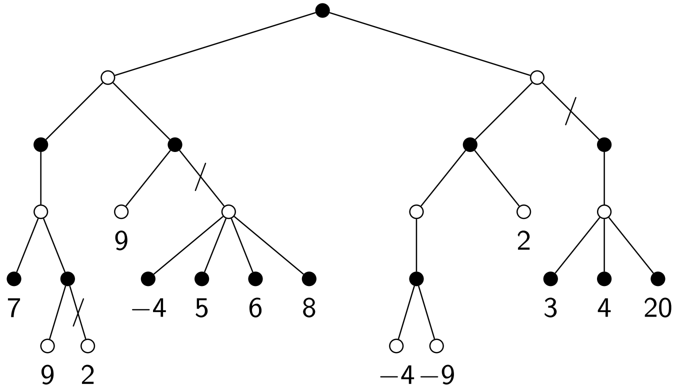
- After exploring a successor of s, if α_s increases beyond β_s, then we need not explore any other successors of s since our adversary will ensure s is never reached.

- If our adversary is the turn-taker at s:
  - $\alpha_s = \alpha_{\text{pt}(s)}$ and will remain fixed.
  - β_s decreases to the minimum value of s's successors as they explored.



- After exploring a successor of s, if β_s decreases beyond α_s, then we need not explore any other successors of s since we will ensure s is never reached.

- Using the min-max strategy with α/β-pruning, we prune the nodes shown below:



Game Strategies: Min-Max w/ α/β-Pruning Psuedocode

```
procedure AlphaBetaSearch(s, t, α, β)
    if s ∈ T then                                        ▷ s is a terminal state
        return r_1(s)
    if t = 1 then
        for a ∈ A(s) do
            α ← max {α, AlphaBetaSearch(S(s, a), 2, α, β)}
            if α ≥ β then
                break
        return α
    if t = 2 then
        for a ∈ A(s) do
            β ← min {α, AlphaBetaSearch(S(s, a), 1, α, β)}
            if α ≥ β then
                break
        return β
```

- The (MinMax) strategy assumes that our adversary always plays perfectly; thus, it maximizes our minimum expected reward.
- This is why it is a good strategy to use if we do not have any other knowledge of our adversary.
- However, in most cases, we have some estimate of $p(\cdot|s)$, whenever the adversary is the turn-taker at s.
- In such cases, it is very much possible for us to have done better.

- In this strategy, we keep an estimate of our adversary's strategy, $\tilde{p}(\cdot|s)$.
- We will study how to compute $\tilde{p}$ later.
- For now, we simply use it in (ExpRwd) and get

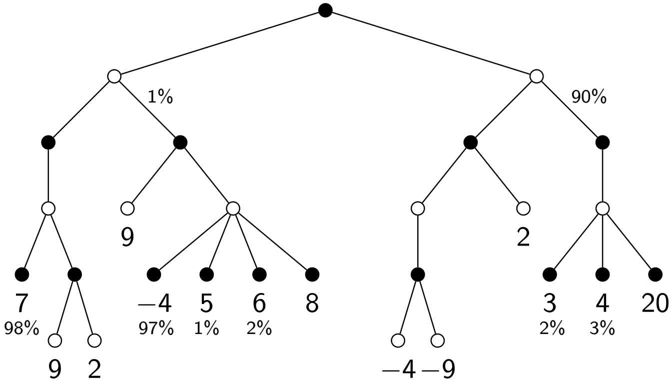
$$v_1(a) = \begin{cases} r_1(a) & S(s,a) \in T \\ \max\limits_{a' \in A(s')} v_1(a') & S(s,a) \notin T \text{ and we are the turn-taker} \\ \sum\limits_{a' \in A(s')} v_1(a')\tilde{p}(a'|s) & S(s,a) \notin T \text{ and we are not the turn-taker} \end{cases} \tag{ExpMax}$$

Game Strategies: ExpectMax Psuedocode

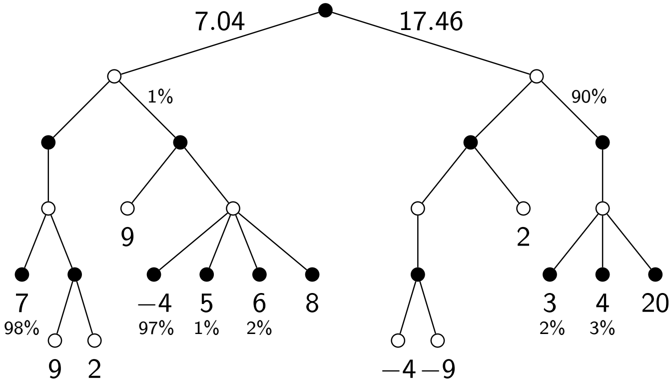
- For any non-terminal circumstance, we recursively compute either the maximum or expected-value of its successors' values.

procedure $\textsc{ExpectMaxSearch}(s, t)$
    if $s \in T$ **then** $\triangleright$ s is a terminal state
        return $r_1(s)$
    if $t = 1$ **then**
        $v_1(s) \leftarrow -\infty$
        for $a \in A(s)$ **do**
            $v_1(s) \leftarrow \max\{v_1(s), \textsc{ExpectMaxSearch}(S(s, a), 2)\}$
    if $t = 2$ **then**
        $v_1(s) \leftarrow 0$
        for $a \in A(s)$ **do**
            $v_1(s) \leftarrow v_1(s) + \tilde{p}(a|s) \times \textsc{ExpectMaxSearch}(S(s, a), 1)$
    return $v_1(s)$

- Suppose $p(\cdot|s)$ was known for all s in which the adversary is the turn-taker:



- We can compute $p(\cdot|s_0)$ as follows:



- A fundamental problem with our approach is that we must explore the entire game-tree before making a decision.
- If we do not have time to explore the entire tree, we need some way to estimate the state values (i.e., a heuristic).
- We can use this estimate in place of the actual expected rewards.

Example: Heuristic for Tic-Tac-Toe Puzzle

- Count the number of ways we could win via our moves thus far.