15-281 Notes
Adversarial Search

Carnegie Mellon University
Artificial Intelligence

Contents

1 Adversarial Search

1.1 Game Playing and Game Types

In adversarial search, we consider scenarios where multiple agents have conflicting goals. These situations arise naturally in competitive games, where one player's gain is another player's loss.

1.1.1 Historical Context

Game-playing AI has a rich history of milestones:

Checkers: In 1950, the first computer player was developed. By 1994, Chinook became the first computer world champion, ending Marion Tinsley's 40-year reign using a complete 8-piece endgame database. In 2007, checkers was completely solved with an endgame database of 39 trillion states.

Chess: Early work began in 1945-1960 with pioneers like Zuse, Wiener, Shannon, Turing, Newell, Simon, and McCarthy. Gradual improvement continued under the "standard model" through the 1960s onward. In 1997, Deep Blue defeated human champion Gary Kasparov in a six-game match, examining 200 million positions per second and extending search up to 40 ply. Current programs running on a PC rate above 3200 Elo (compared to 2870 for Magnus Carlsen).

Go: Zobrist's 1968 program could play legal Go moves, though barely competently given the enormous branching factor ($b > 300$). From 2005-2014, Monte Carlo tree search enabled rapid advances, with programs beating strong amateurs and professionals with handicaps.

1.1.2 Classifying Games

Games can be categorized along several axes:

- **Deterministic** versus **stochastic**: Does randomness play a role?
- **Perfect information** versus **imperfect information**: Can players see the complete game state?
- Number of players: One, two, or more?
- **Turn-taking** versus **simultaneous**: Do players alternate moves or act simultaneously?
- **Zero-sum** versus **general-sum**: Are players' utilities strictly opposed?

1.1.3 Zero-Sum versus General Games

Zero-sum games represent pure competition. Agents have opposite utilities (they sum to zero, so we can formulate the problem based on player one's utility, where player one maximizes while the other minimizes. Examples include chess, checkers, and Go.

General games involve agents with independent utilities. These games can exhibit cooperation, indifference, competition, shifting alliances, or combinations thereof. Examples include many multi-player games and real-world scenarios.

1.1.4 Standard Game Formulation

We focus on **standard games**, which are deterministic, observable, two-player, turn-taking, and zero-sum. These games are formulated as:

- **Initial state**: s_0 represents the starting configuration
- **Players**: Player(s) indicates whose turn it is in state s
- **Actions**: Actions(s) gives available moves for the player at state s
- **Transition model**: Result(s, a) specifies the state resulting from action a in state s

- **Terminal test**: Terminal-Test(s) determines if the game has ended
- **Terminal values**: Utility(s, p) gives the payoff for player p at terminal state s

Our goal is to calculate a **contingent plan**, also called a **strategy** or **policy**, which recommends a move for every possible game state.

1.2 The Minimax Game Trees

The **minimax algorithm** computes optimal play in zero-sum games by assuming both players play optimally. Each game state has a **minimax value**, which represents the best outcome the current player can guarantee with optimal play.

We can represent minimax games as a tree of MAX and MIN nodes, where branches are actions available to the player at that state, leading to the next state where it is the next player's turn.

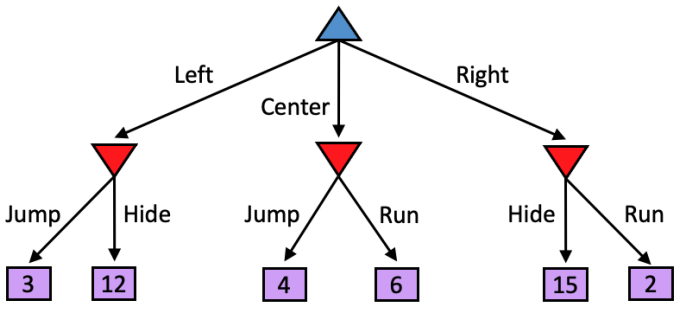
At **MAX nodes** (often represented by blue, upward-pointing triangles), the value is the maximum over the values of the children.

At **MIN nodes** (often represented by red, downward-pointing triangles), the value is the minimum over the values of the children

At **terminal states**, the value is given directly by the utility function. Note that this is always with respect to the player associated with the MAX nodes; (the value of the MIN player is simply the negation of this value).

1.2.1 Example

Consider the following game where the blue player goes first. The blue player is trying to maximize the utilities at the leaves of the tree by choosing action Left, Center, or Right. Then, the red player chooses among its available actions, trying to minimize the values of the resulting state.



Working from the leaves:

- The left MIN node computes $\min(3, 12) = 3$
- The center MIN node computes $\min(4, 6) = 4$
- The right MIN node computes $\min(15, 2) = 2$
- The root MAX node computes $\max(3, 4, 2) = 4$

Therefore, the minimax value at the root is 4... But the number 4 isn't quite the answer we are looking for at the root... We need to select an action! Specifically, the blue MAX player should select the Center action, which leads to this minimax value being 4.

1.2.2 Generic Minimax Pseudocode

The minimax algorithm can be implemented recursively:

```
function max_decision(state):
    return the action in state.actions that gives the max value(state.result(action))

function value(state):
    if state.is_leaf:
        return state.value
    if state.player is MAX:
        return max over value(state.result(a)) for a in state.actions
    if state.player is MIN:
        return min over value(state.result(a)) for a in state.actions
```

1.2.3 Computational Efficiency

Minimax operates similarly to exhaustive depth-first search with the following complexity:

- **Time complexity**: $O(b^m)$ where b is the branching factor and m is the maximum depth
- **Space complexity**: $O(bm)$ when using depth-first exploration

For chess with $b \approx 35$ and $m \approx 100$, exact solution is completely infeasible. This illustrates the concept of **bounded rationality** (Herbert Simon): intelligent agents, including humans, cannot always compute optimal solutions and must make decisions with limited computational resources. More on this in lecture!

1.3 Games with Chance: Expectimax

1.3.1 Motivation

Not all games involve purely adversarial opponents. Some games include elements of chance (dice, card draws), and even in deterministic games, opponents may not play optimally. Making worst-case assumptions about stochastic or suboptimal opponents leads to poor decisions.

1.3.2 Probabilities and Random Variables

A **random variable** represents an event whose outcome is unknown. A **probability distribution** assigns weights to possible outcomes, where probabilities sum to one.

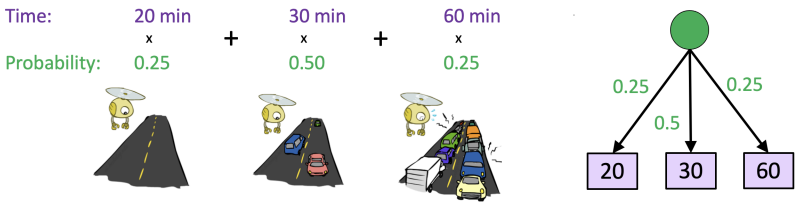
Example: Traffic on the freeway can be modeled as a random variable T with outcomes $T \in \{\text{none}, \text{light}, \text{heavy}\}$ and distribution:

$$P(T = \text{none}) = 0.25$$
$$P(T = \text{light}) = 0.50$$
$$P(T = \text{heavy}) = 0.25$$

1.3.3 Expected Value

The **expected value** of a function of a random variable is the average of the values weighted by their probabilities:

$$\mathbb{E}_X[f(X)] = \sum_x P(X = x) f(x)$$



Example: Let $f(\text{traffic})$ map the Traffic outcome to the time it take to get to the airport:

Traffic (T):	none	light	heavy
Probability:	0.25	0.50	0.25
f(traffic):	20 min	30 min	60 min

Expected value of time over Traffic is:

$$\mathbb{E}_T[f(T)] = \sum_t P(T = t) f(t)$$

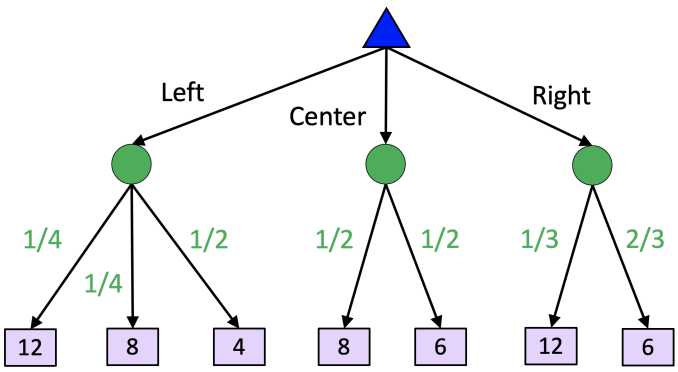
$$= 0.25 \times 20 + 0.50 \times 30 + 0.25 \times 60 = 35 \text{ minutes}$$

1.3.4 Expectimax Algorithm

Expectimax (or expectiminim) extends minimax to handle chance nodes. Instead of assuming the opponent minimizes our utility, we compute expected values over stochastic outcomes.

We can essentially model chance as a player in our game tree. Instead of s' being the deterministic result of taking action a, s' is one of the possible outcomes of the next state. Then, MAX nodes, MIN nodes, and CHANCE (or EXPECTATION) nodes compute values as follows:

- At MAX nodes: $V(s) = \max_a V(s')$
- At MIN nodes: $V(s) = \min_a V(s')$
- At CHANCE nodes: $V(s) = \sum_{s'} P(s')V(s')$



Example: Consider choosing among three actions (Left, Center, Right) where each leads to a chance node with specified probabilities and terminal values. Computing expected values:

- Left: Leads to outcomes with terminal values; expected value = 7
- Center: Leads to outcomes with terminal values; expected value = 7
- Right: Leads to outcomes with terminal values; expected value = 8
- Therefore, the optimal action is Right with expected value 8

The key insight is that we select the action leading to the highest expected value, not the highest individual outcome.

1.3.5 Expectiminimax Pseudocode

Adding a case for an expectation node, we start to have an even more general game tree implementation:

```
function value(state):
    if state.is_leaf:
        return state.value
    if state.player is MAX:
        return max over value(state.result(a)) for a in state.actions
    if state.player is MIN:
        return min over value(state.result(a)) for a in state.actions
    if state.player is CHANCE:
        return sum over P(s)*value(s) for s in state.next_states
```