1 Local Search and Adversarial Search

In previous lectures, the focus was on finding a sequence of actions (a path) to reach a goal (e.g., A* search). However, in many AI problems, the path to the solution is irrelevant; only the final state matters.

Optimization Problem

An **Optimization Problem** is a problem where all states are technically solutions, but they have different qualities. The goal is not to reach a state, but to find the **best state** (Global Optimum) according to an **Objective Function**.

The State Landscape Imagine the state space as a physical landscape where "elevation" corresponds to the value of the objective function.

- **Global Maximum**: The highest peak in the entire state space. The optimal solution.
- **Local Maximum**: A peak that is higher than all its immediate neighbors but lower than the global maximum.
- **Shoulder**: A flat region (plateau) that can be an intermediate step rising to a peak.
- **Flat Local Maximum**: A plateau that is a local maximum.

1.1 Local Search Algorithms

Local search algorithms operate by maintaining a single **Current State** and iteratively moving to neighboring states.

- **Advantages:** Uses very little memory (constant space) and can find reasonable solutions in infinite/massive search spaces.
- **Disadvantages:** No guarantee of completeness (finding a solution) or optimality (finding the best solution).

1.1.1 Hill Climbing (Greedy Local Search)

The Hill Climbing algorithm is a loop that continually moves in the direction of increasing value (uphill).

1. Generate all neighbors of the current state.
2. Move to the neighbor with the highest value.
3. Terminate when no neighbor has a higher value than the current state.

Metaphor

Hill climbing is like "climbing Mount Everest in thick fog with amnesia." You only know the slope under your feet, not where the true peak is.

Problems with Hill Climbing

- **Local Optima**: The algorithm halts at a local peak, missing the higher global peak elsewhere.
- **Ridges**: A sequence of local maxima that is difficult for greedy algorithms to navigate. If the ridge rises diagonally but movement is restricted to cardinal directions (North/East), the algorithm may think it's at a peak because every single step leads downhill, even though the ridge continues up.
- **Plateaus**: Flat areas where the algorithm cannot determine which direction is better, leading to random wandering.

Variants to Solve Local Optima

- **Randomized Restart**: Run the algorithm multiple times from different random starting positions.
- **Stochastic Hill Climbing**: Choose a successor randomly, weighted by the steepness of the uphill move (better moves have higher probability).

1.1.2 Beam Search

Instead of keeping just one state in memory, **Beam Search** keeps track of k states.

- At each step, generate all successors of all k states.
- Select the best k successors from the complete list to be the new current states.
- This is different from k random restarts because the "beams" (threads) can communicate/converge on the most promising area.

1.1.3 Simulated Annealing

This algorithm combines Hill Climbing with a Random Walk to escape local optima. It is inspired by metallurgy (heating metal and cooling it slowly to toughen it).

Mechanism

Instead of always picking the best move, the algorithm sometimes accepts "bad" moves (downhill) to explore the space. The probability of accepting a bad move depends on the **Temperature (T)**.

The Process:

- **High T (Early phase):** The algorithm accepts bad moves frequently (exploring widely, behaving like a random walk).
- **Low T (Late phase):** The algorithm rarely accepts bad moves (exploiting the local area, behaving like hill climbing).

The Probability Formula: If the move improves the situation ($\Delta E > 0$), accept it always. If the move is worse ($\Delta E < 0$), accept it with probability:

$$P = e^{\frac{\Delta E}{T}}$$

As $T \to 0$, the probability of accepting a negative ΔE approaches 0.

1.2 Gradient Descent

When the state space is **continuous** (rather than discrete steps), we use Gradient Descent. This is the foundation of training Neural Networks.

Gradient

The gradient $\nabla J(\theta)$ is a vector of partial derivatives pointing in the direction of the steepest ascent.

To minimize a cost function $J(\theta)$:

$$\theta_{new} = \theta_{old} - \lambda \nabla J(\theta)$$

Where λ is the **Learning Rate**.

1.2.1 The Learning Rate (λ)

This hyperparameter controls the step size.

- **Too Small:** Convergence is extremely slow; might get stuck in local minima.
- **Too Large:** The algorithm may overshoot the minimum, oscillate, or diverge (move away from the solution).

1.3 Brief Note: SAT and Complexity

The Boolean Satisfiability Problem (SAT) is finding an assignment of variables that makes a logical formula true.

- **Cook's Theorem**: SAT is NP-Complete.
- **NP-Complete**: Problems that are effectively "hardest" in NP. If you can solve one NP-Complete problem in polynomial time, you can solve all of them ($P = NP$).
- **GSAT:** A local search algorithm for SAT that flips variable assignments to minimize unsatisfied clauses.

1.4 Adversarial Search (Games)

This domain deals with multi-agent environments where agents have conflicting goals (Zero-Sum Games).

Zero-Sum Game

A situation where one player's gain is exactly the other player's loss. The total utility remains constant (usually 0).

1.4.1 Game Trees

Unlike standard search trees, nodes represent game states, and levels alternate between players:

- **MAX**: The agent trying to maximize the utility value (Us).
- **MIN**: The opponent trying to minimize the utility value (Enemy).

1.4.2 The Minimax Algorithm

The Minimax algorithm determines the optimal move by performing a Depth-First Search (DFS) to the end of the game tree and then reasoning **backwards** (bottom-up).

[Image of minimax game tree example]

The Process:

1. **Dive (DFS):** Traverse the tree all the way down to the leaf nodes (terminal states).
2. **Evaluate:** Assign utility values to the leaf nodes (e.g., Win=1, Loss=-1, or numerical scores).
3. **Propagate (Backtracking):**
   - If the parent is a **MAX** node: It looks at its children and picks the **highest** value.
   - If the parent is a **MIN** node: It looks at its children and picks the **lowest** value.
4. **Root Decision:** When the values reach the root, MAX chooses the move leading to the highest value.

Note: If multiple moves have the same value, standard implementations often pick the one found first (left-most).

Complexity:

- Time: $O(b^m)$ (Exponential). b is branching factor, m is max depth.
- Space: $O(bm)$ (Linear), due to Depth First Search nature.

1.5 Alpha-Beta Pruning

Because Game Trees grow exponentially (10^{40} nodes for Chess), we cannot search the whole tree. **Pruning** allows us to ignore branches that will certainly not be chosen, without affecting the final result.

1.5.1 Concept: Alpha and Beta

We maintain two values during the search that represent the "best guarantees" found so far on the path from the root.

- α **(Alpha)**: The best (highest) value **MAX** can guarantee so far.
  - Initialized to $-\infty$.
  - Updated only at **MAX** nodes (when a child returns a value higher than current α).
- β **(Beta)**: The best (lowest) value **MIN** can guarantee so far.
  - Initialized to $+\infty$.
  - Updated only at **MIN** nodes (when a child returns a value lower than current β).

1.5.2 The Pruning Condition

A branch is pruned (cut off) immediately when:

$$\alpha \geq \beta$$

Why does this work?

- **Beta Cutoff (at MIN node):** If MIN finds a move that results in a value v where $v \leq \alpha$, MIN will choose that (or something even lower). However, the MAX player at the parent node already has a guaranteed option worth α from a previous branch. MAX will therefore **never** choose the path to this MIN node. The rest of MIN's children are irrelevant.
- **Alpha Cutoff (at MAX node):** Symmetrically, if MAX finds a move worth $v \geq \beta$, MIN (at the parent node) will never allow the game to reach this MAX node, because MIN already has a better option (worth β) elsewhere.

Efficiency: With optimal move ordering (checking best moves first), Alpha-Beta pruning effectively doubles the searchable depth, reducing complexity to $O(b^{m/2})$.

1.6 Advanced Game Search

1.6.1 Evaluation Functions

Since we cannot reach terminal nodes in complex games (Chess/Go), we cut off the search at a specific depth and use a **Heuristic Evaluation Function**. This estimates the probability of winning from that state (e.g., Material value in chess: Pawn=1, Queen=9).

1.6.2 Monte Carlo Tree Search (MCTS)

Used in games like Go where branching factors are too high for Minimax ($b \approx 250$).

- Instead of exhaustive search, it plays many random "simulation" games from the current state to the end.
- It builds statistics: "In 80% of random games starting from move A, I won."
- **AlphaGo:** Combined MCTS with Deep Neural Networks (to predict move probabilities and value positions) to defeat the world champion.