FOUNDATIONS OF ALGORITHM DESIGN AND MACHINE LEARNING

DATE: 11 Feb 2025

MIN-MAX SEARCH:

In adversarial two-player games like chess or tic-tac-toe, the challenge isn't just solving the problem—it's also countering an opponent who is actively working against you. This is a common scenario in AI, where you must navigate a situation with an adversary. The goal is to find the best strategy to win by exploring all possible moves and outcomes.

Example: Tic-Tac-Toe

- You are **Player X**, and your opponent is **Player O**.
- When it's your turn, you make a move, and your opponent responds.
- You must think ahead: *If I move here, where will my opponent move next?*
- Evaluate all possible moves your opponent could make and determine the best response.

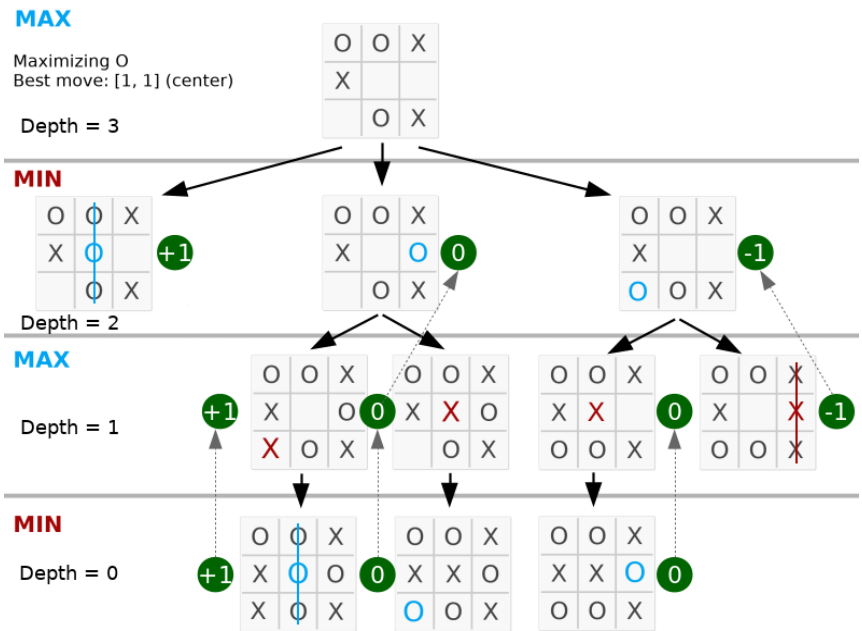
Scenario:

1. You place an **X** in a spot.
2. Your opponent, **O**, can respond by placing their mark in one of several available spots.
3. You evaluate each possibility and choose the move that maximizes your chances of winning.

The Minimax Algorithm:

The Minimax algorithm is a key strategy in AI for solving two-player games. It helps decide the best move by considering both your goals and your opponent's goals. It assumes your opponent will always play optimally to minimize your chances of winning.

Game Tree:



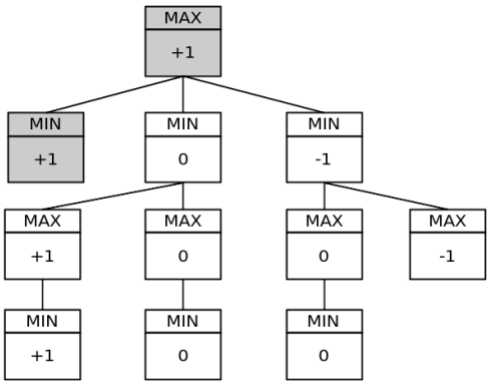
Key Concepts:

1. **Two Players with Opposite Goals**:
    - **Player 1 (Maximizer)**: Aims to maximize their chances of winning.
    - **Player 2 (Minimizer)**: Aims to minimize Player 1's chances (or maximize their own).
2. **Alternating Moves**:
    - On your turn, choose the move with the best outcome.
    - On your opponent's turn, they choose the move that's worst for you (or best for them).
3. **Recursive Search**:
    - The algorithm explores all possible moves and counter-moves, alternating between maximizing and minimizing at each level of the game tree.
    - It continues until it reaches a terminal state (win, lose, or draw).
4. **Scoring**:
    - In zero-sum games (e.g., tic-tac-toe, chess), outcomes are represented as scores:
        - **+1**: You win.
        - **-1**: Your opponent wins.
        - **0**: It's a draw.
    - The algorithm calculates the best move by assuming your opponent will always play optimally to minimize your score.

Why Minimax is Useful:

- It systematically explores all possible moves and counter-moves.
- It prepares you for the worst-case scenario by assuming your opponent plays optimally.
- It's widely used in AI for games like chess, checkers, and tic-tac-toe.

Simplified game tree:



This tree has 11 nodes. The full game tree has large number of nodes.
In a more complex game, such as chess, it's hard to search whole game tree. However, Alpha–beta Pruning is an optimization method to the minimax algorithm that allows us to disregard some branches in the search tree, because it cuts irrelevant nodes (subtrees) in search.

SEARCH APPROACH BASED ON PERTURBING:

Alternative to the iterative approach which we used for solving TSP, we can also find the solution for a problem in the following way:

1. Starting with an initial solution.
2. Perturbing (modifying) the solution in a constructive manner to explore new possibilities.
3. Evaluating the new solutions and repeating the process.

For example, in the **Traveling Salesman Problem (TSP)**, you might start with a route like **A → B → C → D → E → B**. A perturbation could involve swapping two nodes (e.g., swapping **B** and **C**) to create a new route: **A → C → B → D → E → B**. The goal is to iteratively improve the solution by reducing the total cost.

However, if you want to stop in a feasible limit, the challenge is deciding **when to stop** the search. One approach is to monitor the cost:

- If the cost stops decreasing (or starts increasing) after several iterations, you can freeze the solution.

Local Optima:

In optimization problems, a common issue of using the above perturbing-based search approach is getting stuck in **local optima**—solutions that are better than nearby solutions but not the best overall.
To address this:

1. Randomize the starting point: You can use **cross-validation** or **stochastic methods** to explore multiple starting points and avoid getting stuck in local optima. By trying different initial solutions, you increase the chances of finding a better global solution.
This process is like **stochastic gradient descent** in machine learning, where randomization and iterative improvement are key to finding optimal solutions. In minimization problems, gradient descent helps iteratively move toward the best solution by following the steepest decrease in cost. In maximization problems, the opposite approach (called gradient ascent / hill climbing) is used.

2. Simulated Annealing: It is a technique inspired by the physical process of annealing in metallurgy, where a material is heated and slowly cooled to reduce defects and improve its structure.

The Shake Analogy:
Imagine you're trying to climb a rugged terrain to find the lowest point (the global minimum). Initially, you might get stuck in a small valley (local minimum). To escape, you **shake** the system with high energy, allowing it to jump out of the valley and explore other areas. Gradually, you reduce the intensity of the shaking (energy) to stabilize and settle into the lowest possible point.

a) **High Initial Energy:**

- At the start, you apply a lot of energy (shaking) to explore the search space broadly and avoid getting trapped in local optima.
- This is analogous to heating a material to a high temperature, allowing atoms to move freely.

b) **Gradual Reduction in Energy:**

- Over time, you reduce the shaking (energy) to allow the system to settle into a stable, low-energy state.
- This mimics the cooling process in annealing, where the material slowly solidifies into a more ordered structure.

Why Reduce Energy Over Time?

1. Exploration vs. Exploitation:

- High initial energy allows for **exploration** of the search space, helping to escape local optima.
- Reduced energy over time focuses on **exploitation**, refining the solution and stabilizing around the global optimum.

2. Avoiding Overshooting:

- If the energy remains high, the system might keep jumping around and fail to settle into a stable solution.
- Gradually reducing the energy ensures the system converges to the best possible solution

Mathematical Representation:

The energy reduction in simulated annealing is often modelled using the **Boltzmann distribution**, where the energy decreases exponentially over time:

$$E(t) = E_0 \cdot e^{-\alpha t}$$

- $E(t)$: Energy at time t.
- E_0: Initial energy.
- α: Decay constant.
- t: Time or number of iterations.

This exponential decay ensures that the system explores widely at the beginning and fine-tunes its search as it progresses.

GENETIC ALGORITHMS:

Genetic Algorithms (GA) are a powerful **optimization technique** inspired by **natural selection and evolution**. They evolve a population of candidate solutions over multiple generations to find an optimal or near-optimal solution.

Solving the Traveling Salesman Problem (TSP) Using Genetic Algorithm (GA)

1. Introduction to TSP and Genetic Algorithm

The **Traveling Salesman Problem (TSP)** is an optimization problem where a salesman must visit a set of cities exactly **once** and return to the starting city while minimizing the travel distance. Since the problem is **NP-hard**, finding the exact solution for large datasets is computationally expensive.

A **Genetic Algorithm (GA)** is a heuristic search method inspired by **natural selection and evolution**. It is useful for solving optimization problems like TSP by iteratively improving solutions through selection, crossover, and mutation.

2. Representation of TSP in GA

- **Genes:** Each city is represented as a **gene**.
- **Chromosome (Solution Candidate):** A valid tour (permutation of cities) represents a **chromosome**.
- **Population:** A group of chromosomes representing multiple potential solutions.
- **Fitness Function:** The **total path length** of a chromosome (route). **Lower distance = higher fitness**.

3. Genetic Algorithm for TSP

Step 1: Initialization (Creating Initial Population)

- Generate a population of **N** random routes (chromosomes).
- Each route is a random permutation of cities.
- Example for 5 cities:

Route 1: [A → C → E → B → D → A]

Route 2: [B → A → E → D → C → B]

Step 2: Calculate Fitness

- The **fitness function** calculates the **total distance** of a route.
- **Formula:** F=1/Total Distance
    - A **shorter distance** gives a **higher fitness value**.
- Example:

  Route 1: A → C → E → B → D → A, Distance = 120 km

  Route 2: B → A → E → D → C → B, Distance = 140 km
    - **Fitness of Route 1 = 1/120 = 0.0083**
    - **Fitness of Route 2 = 1/140 = 0.0071**
    - Route 1 is **fitter** than Route 2.

Step 3: Selection (Choosing Parents)

- Select **two fittest chromosomes** as parents using methods like:
    - **Roulette Wheel Selection** (probability-based).
    - **Tournament Selection** (best from a subset).
    - **Rank Selection** (higher-ranked individuals have a higher probability).

Step 4: Crossover (Recombining Parents)

- Combine two parent routes to generate a new child route.
- **Example:** Using **Order Crossover (OX)**:

Parent 1: A → B → C → D → E

Parent 2: E → D → C → B → A

Crossover at two points:

- Copy **B → C → D** from Parent 1.
- Fill remaining cities from Parent 2:
    - **Child Route:** E → B → C → D → A

Step 5: Mutation (Introduce Variations)

- Swap two random cities **to avoid local optima**.
- Example:
    - Before Mutation: E → B → C → D → A
    - Swap **B** and **D** → E → D → C → B → A

Step 6: Repeat Until Stopping Condition

- Repeat steps **2 to 5** for a number of **generations** or until a threshold distance is reached.
- **Cooling Mechanism:** A variable controlling the number of iterations to prevent unnecessary computations.

GAs are widely used for **optimization problems**, such as **TSP, scheduling, machine learning, and AI-based problem-solving**

When to Apply Genetic Algorithms:

They work well in problems where solutions can be **decomposed into sub-problems**, solved independently, and then **stitched together** to form a complete solution.

Key Conditions for Applying Genetic Algorithms

1. **Problem Decomposition:**
   - The problem should be divisible into smaller, solvable sub-problems.
   - Example: In combinatorial optimization, you can break down a large problem into smaller components, solve them, and combine the results.

2. **Stitching Solutions**:
   - GA relies on **crossover** and **mutation** to combine and modify partial solutions.
   - Example: In the Traveling Salesman Problem (TSP), you can combine segments of different routes to create a new, potentially better route.

Mostly, Combinatorial optimization relies primarily on these methods - Heuristic search, Game search (Min Max search), Simulated Annealing and Genetic Algorithms.