Artificial Intelligence

An Unofficial Guide to the Georgia Institute of Technology's CS6601

Christian Salas

csalas9@gatech.edu

Last Updated: December 10, 2024

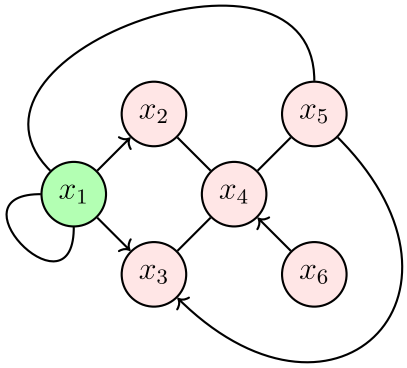
These notes were created with personal effort and dedication. They may contain typos, errors, or incomplete sections. If you find them helpful or wish to provide feedback, feel free to reach out.

Have fun!

Disclaimer: I am a student, not an expert. While I aim for accuracy, there may be mistakes. If you spot any issues, please contribute or contact me.

Contents

1 Search



> *"Algorithms that cannot remember the past are doomed to repeat it."*
> – Chapter 3, Russell and Norvig

1.1 Definition of a Search Problem

A search problem may be broken down into the following parts:

1. **Initial State:** The starting point of the AI.
2. **Actions:** Set of possible actions given a state.

   $actions(s) \rightarrow \{a_1, a_2, a_3\}$
3. **Result:** Resulting state after applying an action.

   $result(s, a) \rightarrow s'$
4. **Goal Test:** Determines whether the AI is in the goal state.

   $test(s) \rightarrow \text{True} \mid \text{False}$
5. **Path Cost:** Total cost of a path from start to end state.

   $path_{cost}(s_0 \rightarrow s_1 \rightarrow s_2) \rightarrow n$
6. **Step Cost:** Cost of a single step.

   $step_{cost}(s, a, s') \rightarrow n$

Algorithm 1 Depth-First Search (DFS)

```
Initialize visited as an empty set
Add start node to visited
for each neighbor in graph[start] do
    if neighbor is not in visited then
        Recursively call DFS on neighbor
    end if
end for
return visited
```

1.2 Depth-First Search (DFS)

Definition: Depth-First Search (DFS) is an algorithm that explores as deeply as possible along each branch before backing up to explore the next branch.

Key Characteristics:

- **Strategy:** Always expands the *deepest* node in the search space first.
- **Backtracking:** Backs up to the next deepest node that has unexplored successors when no further progress can be made.
- **Optimality:** Not cost-optimal—DFS returns the first solution it finds, even if it's not the cheapest.
- **Time Complexity:**
  - **Worst Case:** $O(b^m)$, where b is the branching factor and m is the maximum depth of the search space.
  - **Best Case:** $O(d)$, where d is the depth of the shallowest solution.
- **Space Complexity:**
  - **Recursive Implementation:** $O(m)$, where m is the maximum depth, due to the recursive call stack.
  - **Iterative Implementation:** $O(b \cdot m)$, where b is the branching factor and m is the maximum depth of the search space, accounting for the explicit stack used.

1.3 Depth-Limited Search (DLS)

Definition: Depth-Limited Search (DLS) is a modified version of Depth-First Search (DFS) that imposes a fixed depth limit, l, to prevent infinite exploration in deeply nested or cyclic search spaces.

Algorithm 2 Depth-Limited Search (DLS)

```
Initialize visited as an empty set
Set a depth limit $l$
for each node at depth $\leq l$ do
    Expand node and explore its neighbors
    Add explored nodes to visited
end for
return Fail if no solution is found within depth $l$
```

Key Characteristics:

- **Depth Restriction:** Imposes a depth limit l to restrict the depth of exploration.
- **Leaf Nodes:** Treats all nodes at depth l as leaf nodes and does not expand further.
- **Fallback:** If no solution is found within the limit, the algorithm returns *Fail*.
- **Time Complexity:**
  - **Worst Case:** $O(b^l)$, where b is the branching factor and l is the depth limit.
  - **Best Case:** $O(d)$, where d is the depth of the shallowest solution (if $d \leq l$).
- **Space Complexity:**
  - **Recursive Implementation:** $O(l)$, due to the depth of the recursive call stack.
  - **Iterative Implementation:** $O(b \cdot l)$, accounting for the explicit stack used to track nodes at each level.

Advantages:

- Prevents infinite exploration in graphs with infinite depth or cycles.
- Useful when a maximum solution depth is known in advance.
- Reduces the memory consumption compared to Breadth-First Search (BFS).

Challenges:

- **Choosing Depth Limit:** Selecting an appropriate l can be challenging:
  - If l is too low, the solution may be missed.
  - If l is too high, the algorithm may perform unnecessary computations.
- Not guaranteed to find the shallowest solution unless l is set precisely.

Algorithm 3 Iterative Deepening Search (IDS)

```
for depth limit $l$ from 0 to $max\_depth$ do
    Call Depth-Limited Search (DLS) with depth limit $l$
    if $goal$ is found then
        return $goal$
    end if
end for
return $Fail$ if no solution is found within the maximum depth
```

1.4 Iterative Deepening Search (IDS)

Definition: Iterative Deepening Search (IDS) is an algorithm that combines the space efficiency of Depth-First Search (DFS) with the completeness of Breadth-First Search (BFS). It performs repeated Depth-Limited Searches (DLS) with increasing depth limits until the solution is found.

Key Characteristics:

- **Combination of Strategies:** IDS combines the advantages of DFS (low memory usage) and BFS (completeness).
- **Incremental Depth Limit:** Performs multiple Depth-Limited Searches (DLS) with depth limits starting from 0, incrementing until a solution is found.
- **Optimality:** Ensures optimal solutions are found if all step costs are equal.
- **Time Complexity:**
  - **Worst Case:** $O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution.
  - **Overhead:** IDS repeats nodes at shallower depths, but the overhead is bounded and typically less than twice the cost of a single DFS to depth d.
- **Space Complexity:**
  - $O(d)$, where d is the depth of the shallowest solution. This is because IDS only keeps track of the current path and the depth limit, making it as memory-efficient as DFS.

Applications:

- Ideal for problems where the depth of the solution is unknown.
- Commonly used in scenarios like puzzle-solving and pathfinding.
- Particularly effective when memory resources are limited, and completeness is required.

1.5 Breadth-First Search (BFS)

Definition: Breadth-First Search (BFS) is an uninformed search algorithm that explores all nodes at the current depth level before moving to nodes at the next level. It is particularly effective for finding the shortest path in an unweighted graph or tree.

Algorithm 4 Breadth-First Search (BFS)

```
Initialize queue with the start node
Initialize visited as an empty set
while queue is not empty do
    Dequeue node from the queue
    if node is not in visited then
        Add node to visited
        for each neighbor of node do
            if neighbor is not in visited then
                Enqueue neighbor
            end if
        end for
    end if
end while
return Fail if the goal is not found
```

Key Characteristics:

- **Level-Order Exploration:** BFS explores all nodes at the current depth level before moving deeper.
- **Shortest Path:** Guaranteed to find the shortest path in terms of the number of edges if step costs are equal.
- **Completeness:** BFS is complete—it will find a solution if one exists.
- **Time Complexity:** $O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution.
- **Space Complexity:** $O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution, as BFS stores all nodes at the current depth in memory.

Drawbacks:

- **Memory-intensive:** BFS requires storing all nodes at the current depth level, which can lead to high memory usage in large graphs.
- **Exponential growth:** In graphs with a high branching factor, the memory usage grows exponentially with the depth of the search.

1.6 Uniform-Cost Search (UCS)

Definition: Uniform-Cost Search (UCS) is an uninformed search algorithm that explores nodes based on the lowest path cost. It guarantees finding the least-cost solution in graphs where costs are non-negative. UCS is equivalent to **Dijkstra's algorithm** when applied to graphs.

Key Characteristics:

Algorithm 5 Uniform-Cost Search (UCS)

```
Initialize a priority queue with (cost, start, path)
Initialize visited as an empty set
while priority_queue is not empty do
    Dequeue the element with the lowest cost
    Add node to visited
    if node is the goal then
        return path
    end if
    for each neighbor of node do
        Compute the total cost to neighbor
        if neighbor is not in visited then
            Enqueue (cost, neighbor, updated path) into the priority queue
        end if
    end for
end while
return Fail if no solution is found
```

- **Node Expansion Strategy:** UCS **expands the node with the lowest accumulated path cost**, ensuring the least-cost path to the goal is found.
- **Optimality:** UCS is optimal if all edge costs are non-negative.
- **Completeness:** UCS is complete, meaning it will find a solution if one exists.
- **Time Complexity:**
  - $O(b^{C^*/\epsilon})$, where b is the branching factor, C^* is the cost of the optimal solution, and ϵ is the smallest step cost.
- **Space Complexity:**
  - $O(b^{C^*/\epsilon})$, as UCS must store all nodes in the priority queue and visited set.

1.7 Bidirectional Search

Definition: Bidirectional Search is an algorithm that simultaneously explores the search space from both the start and goal states, expanding nodes alternately in each direction. The search terminates when the two frontiers meet, significantly reducing the search space compared to unidirectional methods.

Algorithm 6 Bidirectional Search

```
Initialize forward_queue with the start node
Initialize backward_queue with the goal node
Initialize forward_visited and backward_visited as empty sets
while forward_queue and backward_queue are not empty do
    Expand the frontier of forward_queue
    if intersection is found with backward_visited then
        return Path Found
    end if
    Expand the frontier of backward_queue
    if intersection is found with forward_visited then
        return Path Found
    end if
end while
return Fail if no intersection is found
```

Key Characteristics:

- **Simultaneous Searches:** Bidirectional Search runs two parallel searches: one from the start node toward the goal, and the other from the goal node toward the start.
- **Meeting Point:** The algorithm terminates when the two search frontiers intersect, combining paths to construct the solution.
- **Completeness:** Guaranteed to find a solution if one exists.
- **Optimality:** If the search uses breadth-first expansion in both directions and all step costs are uniform, it guarantees the shortest path.
- **Time Complexity:** $O(b^{d/2})$, where b is the branching factor and d is the depth of the solution. This is more efficient than unidirectional search ($O(b^d)$).
- **Space Complexity:** $O(b^{d/2})$, as each search stores the frontier and visited nodes, but the depth is halved compared to unidirectional search.

1.8 A* Search

Definition: The A* search algorithm is an informed search technique that finds the shortest path from a start node to a goal node. It combines the cost of reaching a node ($g(n)$) with a heuristic estimate of the cost to reach the goal ($h(n)$), ensuring both efficiency and optimality under the right conditions.

Algorithm 7 A^* Search Algorithm

```
Initialize a priority queue with (f-cost, start, path)
Initialize visited as an empty set
Set $g(start) = 0$ and compute $f(start) = g(start) + h(start)$
while priority_queue is not empty do
    Dequeue the node with the lowest $f$-cost
    if node is the goal then
        return path
    end if
    Add node to visited
    for each neighbor of node do
        Compute $g(neighbor) = g(node) + cost$
        Compute $f(neighbor) = g(neighbor) + h(neighbor)$
        if neighbor is not in visited then
            Enqueue (f-cost, neighbor, updated path) into the priority queue
        end if
    end for
end while
return Fail if no solution is found
```

Key Characteristics:

- **Node Expansion Strategy:** Always expands the node with the lowest total cost $f(n) = g(n) + h(n)$.
- **Completeness:** Guarantees finding a solution if one exists, provided $h(n)$ is admissible.
- **Optimality:** Ensures the shortest path is found if $h(n)$ is admissible and consistent.
- **Time Complexity:**
  - $O(b^d)$ in the worst case, where b is the branching factor and d is the depth of the shallowest solution.
  - Can be faster depending on the quality of $h(n)$.
- **Space Complexity:**
  - $O(b^d)$, as A^* must maintain all nodes in memory for its priority queue.

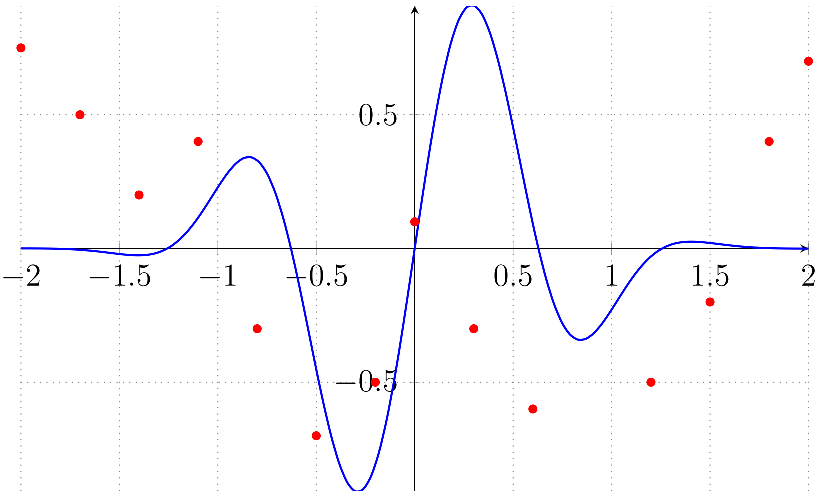
Conditions for Optimality:

- **Admissibility:** The heuristic $h(n)$ must never overestimate the true cost to the goal:
  $$h(n) \leq \text{true cost to goal from } n.$$
- **Consistency:** The heuristic must satisfy the triangle inequality:
  $$h(n) \leq c(n, n') + h(n'),$$
  where $c(n, n')$ is the cost between nodes n and n'.

1.9 Other Informed Search Algorithms

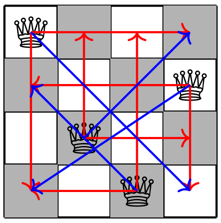
- **Greedy Best-First Search:** Expands nodes with the smallest $h(n)$; efficient but not guaranteed to be optimal.
- **Bidirectional A*:** Enhances A* by searching from both **start and goal**
- **IDA*:** Iterative deepening version of A* that reduces memory usage.
- **RBFS and SMA*:** Memory-bounded alternatives to A* for large problems.
- **Beam Search:** Limits the frontier size for faster but suboptimal results.
- **Weighted A*:** Prioritizes goal-directed paths at the cost of guaranteed optimality.

2 Simulated Annealing



> *"Sometimes letting go and embracing randomness is the key to the solution.."*
>
> – ?

2.1 N-Queens



The N-Queens problem involves placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other. Queens can attack horizontally, vertically, and diagonally. The goal is to find an arrangement where none of these attacks occur.

Brute-Force Approach

In a brute-force approach, we try every possible placement of N queens on the board. For example, on a 4×4 board, there are:

$$16 \times 15 \times 14 \times 13 = \textbf{43,680} \quad \textbf{possible solutions.}$$

This number grows exponentially as N increases, making the brute-force approach impractical for large N.

Optimization and Heuristics

Backtracking Algorithm:

1. Place queens row by row.

2. Check if the current placement is valid (no queens threaten each other).

3. If valid, move to the next row.
4. If no valid placement exists, backtrack to the previous row and try a different position.

Heuristic Methods:

- **Simulated Annealing:** Randomly explores the solution space, occasionally accepting worse placements to avoid getting stuck in suboptimal solutions.
- **Genetic Algorithms:** Uses evolutionary principles like selection, crossover, and mutation to refine queen placements over multiple generations.

2.2 Hill Climbing Algorithm

Hill Climbing is a local search heuristic used to find an optimal solution by iteratively making small changes to the current state. It aims to reach the highest possible point (maximum value) in the search space by moving towards neighboring states that improve the current solution.

- **Algorithm Behavior:** At each step, the algorithm evaluates neighboring states and chooses the one that improves the current solution.
- **Analogy:** The process is analogous to climbing a hill, where the algorithm always chooses the upward path in the hopes of reaching the peak.
- **Drawback:** Hill climbing can get stuck in local maxima, which are points that are higher than neighboring points but not the highest in the entire search space. *This is similar to climbing a mountain and reaching a peak that is not the highest*, with no way to know it's not the global maximum.

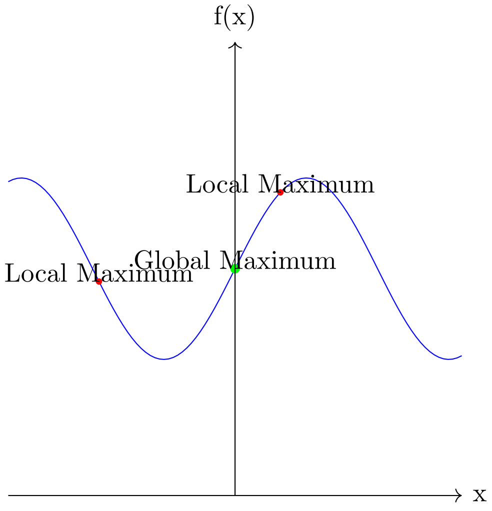


Figure 1: Illustration of Local Maxima and Global Maximum in a Hill Climbing Algorithm. The algorithm may get stuck at one of the local maxima, even though the global maxima is higher.

2.2.1 Random Restart

Random Restart is a technique used to overcome the issue of getting stuck in local maxima. The approach involves:

- **Restarting** the Hill Climbing algorithm from a new random state whenever it reaches a local maximum.
- By performing **multiple restarts**, the algorithm has a higher chance of finding the global maximum, as different starting points may lead to better solutions.

2.3 Simulated Annealing

Simulated Annealing is a probabilistic optimization algorithm inspired by the physical process of heating and then slowly cooling a material to find its optimal configuration (i.e., the lowest energy state). It is particularly useful for problems with a large search space that may contain many local minima.

- **Heating (High Temperature):** During the initial stages, the algorithm explores a wide range of possible solutions, including less optimal ones. This is similar to the high temperature phase, where molecules are more likely to move freely.
- **Cooling (Low Temperature):** As the temperature decreases, the algorithm becomes more selective, refining the best solution found so far, similar to how cooling solidifies molten metal into its final form.
- **Acceptance of Worse Solutions:** To avoid getting stuck in local optima, the algorithm allows for worse solutions with a certain probability, which decreases as the temperature cools. This probability is governed by:

  $$P(\text{accept}) = \exp\left(\frac{\Delta E}{T}\right)$$

  where ΔE is the change in energy (or cost), and T is the current temperature.
- **Cooling Schedule:** The rate at which the temperature decreases is crucial. Common cooling schedules include:

  $$T_{n+1} = \alpha T_n$$

  where α is a constant between 0 and 1, and T_n is the temperature at iteration n. A slower cooling schedule allows for better exploration of the solution space.

The Simulated Annealing algorithm proceeds as follows:

Algorithm 8 Simulated Annealing

```
Initialize *current* as the initial state
Set the initial temperature $T$
while $T >$ *minimum temperature* do
    Pick a random neighboring solution *next*
    Compute $\Delta E =$ *value(next) – value(current)*
    if $\Delta E > 0$ then
        Accept *next* as the current state
    else if $\exp(\Delta E/T) >$ *random(0, 1)* then
        Accept *next* as the current state
    end if
    Decrease the temperature according to the cooling schedule
end while
return *current*
```

2.3.1 Cooling Schedules

The cooling schedule is one of the most important parameters of Simulated Annealing, as it determines how quickly the algorithm ”settles” into a solution. A few common cooling schedules are:

- **Exponential Cooling:** The temperature decreases exponentially at each iteration:

$$T_{n+1} = \alpha T_n, \quad \alpha \in (0, 1)$$

- **Logarithmic Cooling:** The temperature decreases according to a logarithmic function:

$$T_{n+1} = \frac{T_0}{\log(1 + n)}$$

- **Linear Cooling:** The temperature decreases linearly:

$$T_{n+1} = T_n - \Delta T$$

2.3.2 Stopping Criteria

Simulated Annealing can stop based on various criteria, including:

- **Fixed Number of Iterations:** The algorithm runs for a predefined number of iterations.
- **Temperature Threshold:** The algorithm stops when the temperature falls below a certain threshold.
- **No Improvement:** The algorithm stops when no significant improvement has been made after a certain number of iterations.

Advantages:

- Can escape local optima due to the probabilistic acceptance of worse solutions.
- Effective for large and complex search spaces.

Disadvantages:

- The cooling schedule needs to be carefully chosen.
- Computationally expensive for large problems.
- The algorithm may still get stuck in suboptimal solutions if the temperature decays too quickly.

2.3.3 Applications of Simulated Annealing

Simulated Annealing has been successfully applied to a variety of optimization problems, including:

- Traveling Salesman Problem (TSP)
- Job scheduling
- Circuit design
- Neural network training

2.4 Local Beam Search

Local Beam Search is a heuristic search algorithm that maintains multiple states (beams) at once, exploring several paths simultaneously to avoid getting stuck in local maxima.

Key Concepts

- **Beam Width:** The number of states kept at each step.
- **Selection:** At each step, the best k states are selected from the neighbors of the current states.
- **Exploration:** Explores multiple paths to improve the chances of finding a better solution.

Algorithm

Algorithm 9 Local Beam Search

```
Initialize $k$ random states
while not a goal state do
    Generate neighbors of the current states
    Select the best $k$ states
end while
return the best state
```

Advantages and Disadvantages

Advantages:

- Can explore multiple paths simultaneously, reducing the risk of getting stuck in local maxima.

Disadvantages:

- Computationally expensive with large beam widths.
- Still prone to local maxima depending on the beam width and evaluation function.

Applications

Effective for large search spaces in problems like machine learning, combinatorial optimization, and game-playing algorithms.

2.5 Genetic Algorithms

Genetic Algorithms (GAs) are a class of optimization algorithms inspired by the process of natural selection in biology. They use mechanisms such as **selection**, **crossover** (breeding), and **mutation** to evolve a population of candidate solutions toward an optimal solution.

Key Concepts

- **Population:** A set of candidate solutions, often represented as chromosomes (or strings).
- **Selection:** The process of choosing individuals from the population based on their fitness (how good the solution is). This is similar to natural selection.
- **Crossover (Breeding):** The process of combining parts of two or more individuals to create offspring. This simulates reproduction and introduces diversity into the population.
- **Mutation:** Random changes in an individual's genetic code to introduce variability and prevent premature convergence.
- **Fitness Function:** A function used to evaluate how well a solution performs with respect to the problem.

Genetic Algorithm Steps

The typical steps for a genetic algorithm are as follows:

1. **Initialize the Population:** Start with a randomly generated population of candidate solutions.
2. **Selection:** Evaluate the fitness of individuals and select the best solutions to reproduce.
3. **Crossover:** Mate selected individuals to produce offspring, combining their genes.

4. **Mutation:** Introduce small random changes to the offspring to maintain diversity.
5. **Repeat:** Repeat the selection, crossover, and mutation steps for multiple generations until a stopping condition is met (e.g., a maximum number of generations or a satisfactory fitness level).

Example: Solving the Knapsack Problem

Consider a simple **Knapsack Problem** where the goal is to select items to maximize the total value without exceeding a weight limit. Here's how a genetic algorithm can be applied:

- Each individual in the population represents a set of items, where each item is either **included** or **excluded** from the knapsack. This is represented by a binary string, e.g., `10101`, where each bit corresponds to an item.
- The fitness function evaluates how valuable the selected items are, while ensuring the total weight does not exceed the limit.
- In the crossover step, two parents (solutions) could exchange part of their binary string to create offspring. For instance, the two parents `10101` and `11001` might produce the offspring `11101`.
- Mutation might randomly flip some bits, for example, `11101` might mutate to `11001`.
- The process repeats for several generations until an optimal or near-optimal solution is found.

Advantages and Disadvantages

Advantages:

- Can search a large solution space efficiently.
- Suitable for problems with complex, non-linear, or poorly understood search spaces.
- Can escape local optima by using crossover and mutation to explore a wide solution space.

Disadvantages:

- May require many generations to converge to a solution.
- The algorithm's performance is highly dependent on the choice of parameters (e.g., population size, mutation rate).
- Can be computationally expensive for large problems.

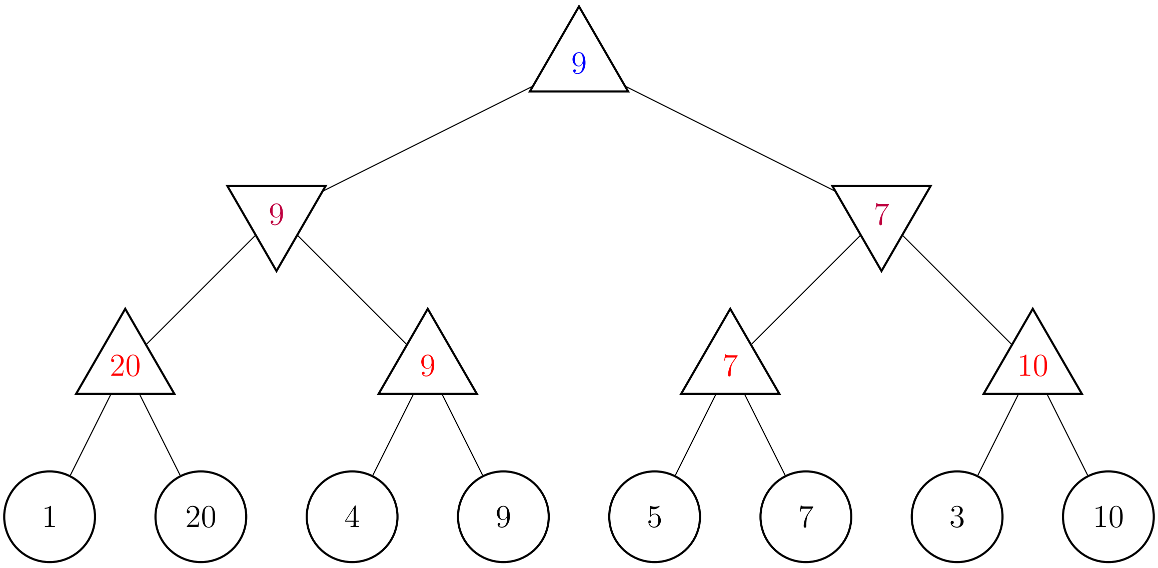
Applications of Genetic Algorithms

Genetic algorithms are widely used in optimization problems, such as:

- Combinatorial optimization (e.g., Traveling Salesman Problem, Knapsack Problem).
- Machine learning (e.g., neural network training, feature selection).
- Game strategy development.
- Engineering design problems.

3 Game Theory

> *"Game theory is the study of mathematical models of conflict and cooperation between intelligent rational decision-makers."*
>
> – Roger B. Myerson



3.1 Minimax Algorithm

The **Minimax Algorithm** is used for decision making in two-player games. It operates by building a game tree where each node represents a possible game state, and the edges represent the moves between these states. The algorithm alternates between the two players' turns (maximizing and minimizing players), with each player trying to maximize their own chances of winning while minimizing the opponent's chances.

Key Concepts

- **Maximizing Player:** The player trying to maximize their score (typically the AI).
- **Minimizing Player:** The opponent trying to minimize the AI's score.
- **Game Tree:** A tree representation of the possible moves in the game, where leaves represent final game outcomes (win, lose, draw).
- **Depth of the Tree:** The number of levels in the game tree that corresponds to how many moves ahead the algorithm will evaluate.

Minimax Algorithm Steps

The algorithm works recursively to evaluate each node in the game tree. Starting from the leaves (final game states), it propagates values up the tree, alternating between the maximizing and minimizing players:

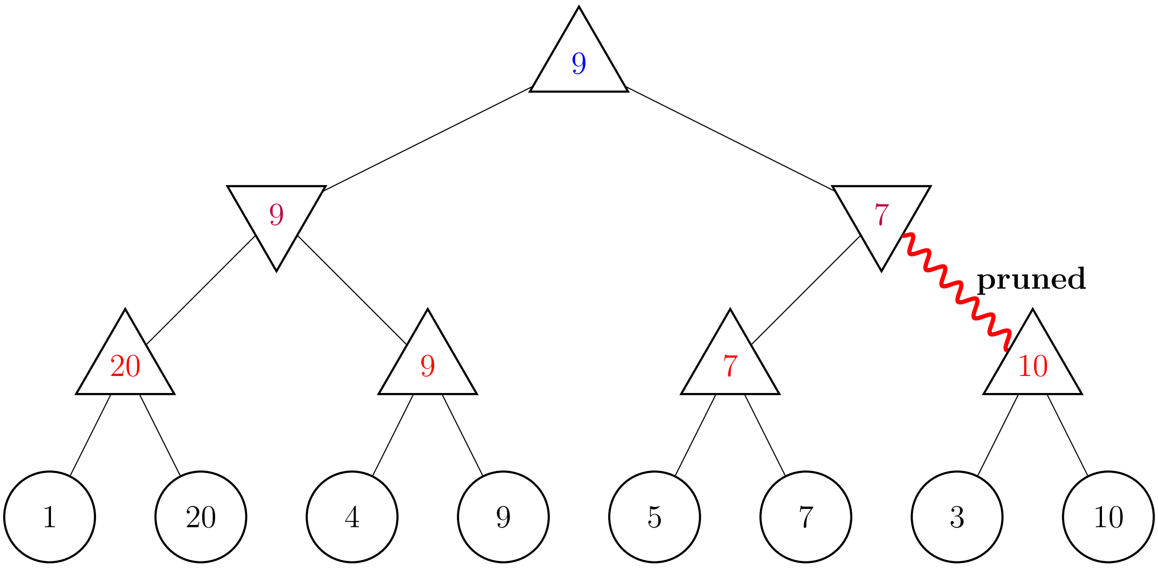
1. **Generate Game Tree:** Construct the game tree with possible moves and outcomes.

2. **Evaluate Leaf Nodes:** Assign values to the leaf nodes based on the game outcome (e.g., win = +1, loss = -1, draw = 0).

3. **Propagate Values Up the Tree:**

   - For maximizing player nodes, select the maximum value from child nodes.
   - For minimizing player nodes, select the minimum value from child nodes.

4. **Choose Optimal Move:** Once the root node is evaluated, the best move is selected by choosing the child node with the optimal value.



3.2 Alpha-Beta Pruning Algorithm

The **Alpha-Beta Pruning Algorithm** is an optimization technique for the minimax algorithm. It reduces the number of nodes that need to be evaluated by cutting off branches that cannot affect the final decision.

Key Concepts

- **Alpha (α):** The best value the maximizing player can guarantee so far (initially $-\infty$).
- **Beta (β):** The best value the minimizing player can guarantee so far (initially $+\infty$).
- **Pruning:** Cutting off branches that will not influence the final decision.

Alpha-Beta Pruning Steps

The algorithm works similarly to minimax, but it maintains two values, α and β, that track the best options for both players:

1. **Initialize Values:** Set $\alpha = -\infty$ and $\beta = +\infty$.

2. **Traverse the Game Tree:** Perform a depth-first search.

3. **Apply the Rules:**
    - At a **Max Node**:
        - Update $\alpha = \max(\alpha, \text{current value})$.
        - If $\alpha \geq \beta$, prune the remaining branches.
    - At a **Min Node**:
        - Update $\beta = \min(\beta, \text{current value})$.
        - If $\beta \leq \alpha$, prune the remaining branches.
4. **Choose Optimal Move:** After evaluating the root, choose the move leading to the best outcome.

Why Use Alpha-Beta Pruning?

- **Efficiency:** Reduces the number of nodes explored.
- **Optimality:** The algorithm still finds the optimal move.
- **Time Complexity:** With perfect pruning, the time complexity is $O(b^{d/2})$, where b is the branching factor and d is the depth.

3.3 Evaluation Function

The **Evaluation Function** estimates the desirability of a game state when the search cannot proceed to terminal states. **It assigns a numerical score based on the current board configuration**.

- At the **leaf nodes**, the game state is evaluated using a heuristic or utility function.
- **Positive values** indicate an advantage for the **maximizing player**, while **negative values** favor the **minimizing player**.
- The evaluation function considers important factors such as:
    - **Material Advantage:** Difference in key game pieces.
    - **Positional Strength:** Control of important areas in the game environment.
    - **Mobility and Potential:** Ability to make future moves effectively.

3.4 Quiescence Search

Quiescence Search is an extension of the minimax search that addresses the horizon effect by exploring unstable positions beyond the standard search depth. It helps avoid evaluating volatile game states prematurely.

Why Use Quiescence Search?

In many games, certain positions are unstable due to tactical possibilities like captures or checks in chess. Evaluating such positions can lead to incorrect conclusions if the search is abruptly stopped. Quiescence search extends the search depth until a stable, ”quiet” position is reached.

How It Works

1. After reaching a leaf node in the regular search, check if the position is **quiet** (stable).
2. If the position is unstable:
   - Expand the game tree by considering only certain **tactical moves** (e.g., captures, checks).
   - Continue searching until a quiet position is reached.
3. Evaluate the resulting quiet position using the evaluation function.

Key Considerations

- **Selective Moves:** Only tactical moves are considered to limit the search space.
- **Search Depth:** Quiescence search is not unlimited—it uses a reasonable depth to prevent infinite search loops.
- **Performance Trade-Off:** While improving accuracy, quiescence search can increase computation time if not properly managed.

3.5 Horizon Effect

The **Horizon Effect** occurs when a game-playing algorithm cannot see far enough ahead due to a limited search depth. As a result, the algorithm may miss critical future consequences, leading to suboptimal decisions.

Description

The search process is constrained by a fixed depth, creating a "horizon" beyond which the algorithm cannot evaluate moves. This limitation can cause the algorithm to delay inevitable losses or fail to anticipate long-term advantages.

Example: Chess

In chess, the horizon effect can occur when:

- A critical piece is at risk of being captured in a few moves, but the search depth is too shallow to detect this threat.
- The algorithm delays the capture by making irrelevant moves, falsely believing it has avoided the loss.

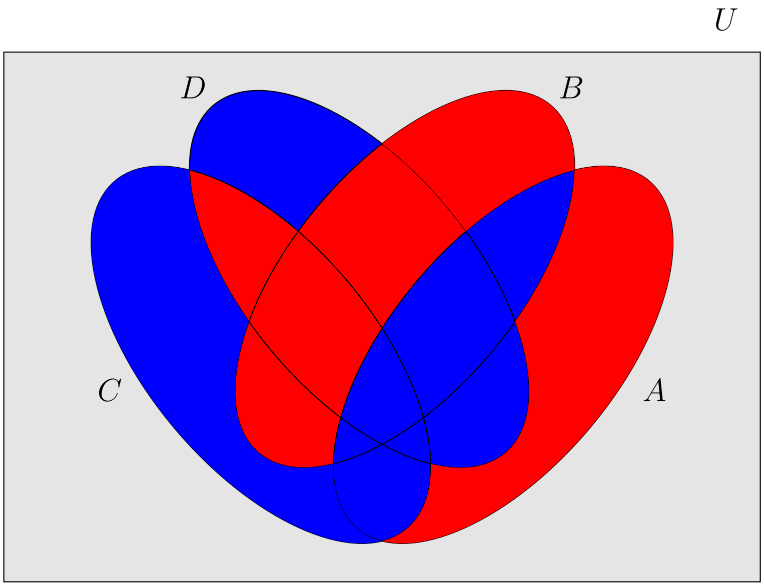
Impact and Mitigation

- **Impact:** The algorithm may:
  - Make short-sighted moves.
  - Miss winning tactics or fail to avoid losing positions.

- **Mitigation Techniques:**
  - **Quiescence Search:** Extend the search beyond unstable nodes to find more accurate evaluations.
  - **Iterative Deepening:** Increase search depth gradually, using earlier results to guide deeper searches.
  - **Selective Search:** Focus the search on promising or dangerous moves to reduce computational load.

4 Constraint Satisfaction Problems

> *"A constraint satisfaction problem is not about finding 'a' solution; it's about finding 'the' solution that fits within a defined space of possibilities."*
>
> – Edward Tsang



Constraint Satisfaction Problems (CSPs) are mathematical problems defined by a set of **variables, domains, and constraints**. The goal is to assign values to variables while satisfying all constraints.

4.1 Components of a CSP

- **Variables:** The set of elements that need to be assigned values (e.g., $X_1, X_2, \ldots, X_n$).
- **Domains:** The set of possible values each variable can take (e.g., $D_1, D_2, \ldots, D_n$).
- **Constraints:** Rules or relationships between variables that must be satisfied (e.g., $X_1 \neq X_2$).

4.2 Types of CSPs

- **Discrete CSPs:** Finite domains (e.g., Sudoku, scheduling).
- **Continuous CSPs:** Infinite domains (e.g., scheduling with continuous time).
- **Binary CSPs:** Constraints involve pairs of variables.
- **Higher-order CSPs:** Constraints involve more than two variables.

4.3 Solving CSPs

Solving a CSP involves finding an assignment of values to variables that satisfies all constraints. This can be done using several methods:

Backtracking Search

- A depth-first search that assigns values to variables one at a time.
- If a constraint is violated, the algorithm backtracks to try a different assignment.

Constraint Propagation

Constraint propagation involves enforcing constraints locally to reduce the search space by eliminating inconsistent values. Common types of consistencies include:

- **Node Consistency:**
  - A variable X_i is **node consistent** if all values in its domain D_i satisfy its unary constraints $C(X_i)$.
  - **Mathematical Definition:**

    $$\forall x \in D_i,\ C(X_i = x) \text{ is true}$$
- **Arc Consistency (AC-3):**
  - A variable X_i is **arc consistent** with another variable X_j if every value in D_i has a corresponding value in D_j that satisfies the binary constraint $C(X_i, X_j)$.
  - **Mathematical Definition:**

    $$\forall x \in D_i,\ \exists y \in D_j \text{ such that } C(X_i = x, X_j = y) \text{ is true}$$
  - AC-3 is an algorithm that iteratively enforces arc consistency by examining each arc (X_i, X_j) and removing inconsistent values from D_i.
- **Path Consistency:**
  - A pair of variables (X_i, X_j) is **path consistent** concerning a third variable X_k if, for every assignment of X_i and X_j, there exists a consistent value of X_k satisfying all relevant constraints.
  - **Mathematical Definition:**

    $$\forall (x_i, x_j) \in D_i \times D_j,\ \exists x_k \in D_k \text{ such that } C(X_i = x_i, X_k = x_k) \wedge C(X_j = x_j, X_k = x_k)$$
  - Path consistency extends arc consistency by considering indirect relationships between variables.

Heuristics for CSPs

- **Minimum Remaining Values (MRV):** Choose the variable with the fewest legal values.
- **Degree Heuristic:** Choose the variable involved in the most constraints.
- **Least Constraining Value:** Choose the value that rules out the fewest choices for other variables.

5 Probability: Understanding Uncertainty

5.1 Random Variables

A **random variable** is a variable whose possible values depend on the outcome of a random process.

1. **Discrete Random Variables:** Take on a finite or countable set of values (e.g., the roll of a die).

$$X = \{1, 2, 3, 4, 5, 6\}, \quad P(X = x) = \frac{1}{6}, \forall x \in X$$

2. **Continuous Random Variables:** Take on values in a continuous range (e.g., the height of individuals).

$$P(X = x) = 0, \quad \text{but } P(a \leq X \leq b) = \int_a^b f_X(x)\,dx$$

5.2 Propositions and Syntax

Propositions are statements about events that can be **true** or **false**.

- $P(A)$: Probability of event A occurring.
- $P(A \cap B)$: Probability that both A and B occur.
- $P(A \cup B)$: Probability that A or B (or both) occur.
- $P(A^c)$: Probability that A does not occur (complement of A).
- **Prior Distribution:** A prior distribution represents the probability of an event or parameter before observing any data or evidence. It reflects initial beliefs about the event based on previous knowledge or assumptions.

  $$P(\theta) \quad \text{where} \quad \theta \text{ is the unknown parameter}$$

  In Bayesian inference, the prior is updated with new evidence using Bayes' theorem:

  $$P(\theta|\text{data}) = \frac{P(\text{data}|\theta)P(\theta)}{P(\text{data})}$$

- **Posterior Distribution:** The posterior distribution **represents the updated probability** of an event or parameter after observing new data or evidence. It combines the prior distribution with the likelihood of the observed data:

5.3 Prior Probability

Prior probability represents the initial belief about an event before observing any evidence. **Example**: The probability that a randomly chosen student owns a pet:

$$P(\text{Pet Owner}) = 0.6$$

5.4 Gaussian Distribution (Normal Distribution):

For continuous random variables, probabilities are defined over intervals using a **probability density function (PDF)**. The Gaussian density function is defined as:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Where:

- μ: Mean (center of the distribution)
- σ^2: Variance (spread of the distribution)

Example: Heights of people in a population often follow a Gaussian distribution.

5.5 Conditional Probability

The probability of event A given that event B has occurred is defined as:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \quad P(B) > 0$$

Example: Probability of liking tea (T) given it's a cold day (C):

$$P(T \mid C)$$

5.6 Inference by Enumeration

To calculate probabilities over multiple variables:

$$P(A) = \sum_{B} P(A \cap B)$$

Example: The probability of passing a test $P(\text{Pass})$ might depend on studying (S) or not:

$$P(\text{Pass}) = P(\text{Pass} \mid S)P(S) + P(\text{Pass} \mid S^c)P(S^c)$$

5.7 Independence

Two events A and B are independent if:

$$P(A \cap B) = P(A)P(B)$$

Example: Flipping two coins. The outcome of one does not affect the other.

5.8 Conditional Independence

Two events A and B are conditionally independent given C if:

$$P(A \cap B \mid C) = P(A \mid C)P(B \mid C)$$

Example: Whether it rains today (R) and whether you take your umbrella (U) might be independent given that you check the weather forecast (F).

5.9 Normalization

Probabilities must sum to 1:

$$\sum_i P(X = x_i) = 1, \quad \text{for discrete variables.}$$

For continuous variables:

$$\int_{-\infty}^{\infty} f_X(x)\, dx = 1$$

6 Bayes Nets

> *"Bayesian networks provide a framework for reasoning under uncertainty by combining probabilistic inference with a graphical representation of dependencies."*
>
> – Daphne Koller

6.1 Bayes Rule

Bayes Rule is a fundamental formula in probability that allows us to update our beliefs about an event based on new evidence. It is given by:

$$P(a \mid b) = \frac{P(b \mid a) \cdot P(a)}{P(b)}$$

Where:

- $P(a \mid b)$: Posterior probability of a given b (updated belief after observing evidence b).
- $P(b \mid a)$: Likelihood of evidence b occurring if a is true.
- $P(a)$: Prior probability of a (belief about a before observing evidence).
- $P(b)$: Marginal probability of b (total probability of evidence b across all possible causes).

6.2 Example: Alien Detection

Imagine you are the captain of the spaceship *Intergalactic Voyager*, exploring deep space. Your ship's sensors are designed to detect alien ships in nearby regions of the galaxy. However, the sensors are imperfect.

- A: Event that there is an alien ship nearby.
- S: Event that the sensor detects alien signals.

Your mission is to calculate the probability of aliens being nearby $P(A \mid S)$ given that the sensors have detected a signal.

Known Information:

- $P(A) = 0.02$: The prior probability of an alien ship being nearby (aliens are rare!).
- $P(S \mid A) = 0.9$: The likelihood that the sensor detects signals when aliens are present.
- $P(S \mid A^c) = 0.1$: The likelihood that the sensor detects signals even when there are no aliens (false positive rate).
- $P(A^c) = 0.98$: The prior probability of no aliens nearby.

Step 1: Total Probability of Sensor Detection

We first calculate the total probability of the sensor detecting a signal, $P(S)$:

$$P(S) = P(S \mid A) \cdot P(A) + P(S \mid A^c) \cdot P(A^c)$$

Substitute the values:

$$P(S) = (0.9 \cdot 0.02) + (0.1 \cdot 0.98) = 0.018 + 0.098 = 0.116$$

Step 2: Posterior Probability of Aliens Nearby

Now, we use Bayes Rule to calculate $P(A \mid S)$:

$$P(A \mid S) = \frac{P(S \mid A) \cdot P(A)}{P(S)}$$

Substitute the values:

$$P(A \mid S) = \frac{0.9 \cdot 0.02}{0.116} \approx 0.155$$

Even with a positive signal, there is only a 15.5% chance that an alien ship is nearby due to the rarity of alien encounters.

6.3 Bayesian Network: Galactic Spy Network

You decide to build a Bayesian Network to better understand the relationships between variables.

Scenario: Alien Presence, Sensor Signal, and Defensive Readiness
Your network includes:

- A: Presence of alien ships.
- S: Sensor signal detection.
- D: Defensive readiness (activated if signals suggest aliens).

Graph Structure:

$$A \to S \to D$$

The **Conditional Probability Tables (CPTs)** might look like this:

$P(A) =$

Alien Presence (A)	$P(A)$
Yes	0.02
No	0.98

$P(S \mid A) =$

A	$S =$ Yes	$S =$ No
Yes	0.9	0.1
No	0.1	0.9

$P(D \mid S) =$

S	$D =$ Ready	$D =$ Not Ready
Yes	0.8	0.2
No	0.1	0.9

6.4 Reasoning in the Network

Given a positive signal (S = Yes), calculate the probability of being ready for defense $P(D = \text{Ready} \mid S = \text{Yes})$:

$$P(D = \text{Ready} \mid S = \text{Yes}) = 0.8$$

Bayesian networks like this help the *Intergalactic Voyager* crew make informed decisions under uncertainty, improving their survival in hostile galaxies!

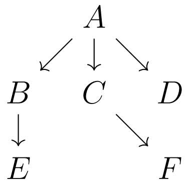
6.5 Bayesian Network

A Bayesian Network is a **probabilistic graphical model** that represents a set of variables and their conditional dependencies using a directed acyclic graph (DAG).

- **Nodes**: Represent random variables.
- **Edges**: Directed edges represent direct dependencies.

6.6 Joint Distribution

Definition: The probability of two or more random variables happening **at the same time**.



$$P(R, S, W) = P(R) \cdot P(S \mid R) \cdot P(W \mid R, S)$$

Example:

- $A = 1$, $B = 2$, $C = 2$, $D = 2$, $E = 2$, $F = 4$

6.7 D-Separation

Definition: A path is **blocked** if information **cannot flow** between two variables, meaning they are independent. A path is **open** if information **can flow**, meaning they are dependent.

6.8 Enumeration Methods

Bayesian Network inference can be done through **two** main enumeration techniques:

6.8.1 Full Enumeration:

- Computes exact probabilities by summing over all possible values of hidden (unobserved) variables.
- Guarantees precise results but becomes **computationally expensive** as the network size increases due to the exponential number of states.
- **Mathematical Formulation:** Given a query $P(X|e)$, where X is the query variable and e is the evidence, the full enumeration process computes:

  $$P(X|e) = \frac{P(X,e)}{P(e)} = \frac{\sum_Y P(X,Y,e)}{\sum_X \sum_Y P(X,Y,e)}$$

  where Y represents all hidden variables.
- **Example:** In a weather prediction network, summing over all possible weather conditions to compute the probability of rain given cloudy skies.
- **Best For:** Small networks or scenarios where exact results are critical.

6.8.2 Optimized Enumeration (Variable Elimination):

- Reduces computation by eliminating hidden variables one at a time, combining and simplifying factors.
- Exploits **conditional independencies** in the network to avoid redundant calculations.
- **Mathematical Process:**

  $$P(X|e) = \frac{1}{Z} \prod_{f \in \text{Factors}} f$$

  where factors are created and reduced through marginalization:

  $$f'(Y) = \sum_Z f(Y,Z)$$
- **Example:** In a disease diagnosis network, removing intermediate variables like symptoms by combining factors to focus on the relationship between evidence and the disease.
- **Best For:** Larger networks where computational efficiency is important.

6.9 Sampling Techniques

When exact inference in Bayesian Networks becomes computationally infeasible, sampling-based approaches offer approximate solutions by generating and evaluating samples.

Overview of Sampling Techniques

- **Rejection Sampling:**
  - Generates samples from the prior distribution $P(X_1, X_2, \ldots, X_n)$.
  - Rejects samples that are inconsistent with the observed evidence e.
  - **Estimated Probability:**
  $$P(X|e) \approx \frac{\text{Count of samples consistent with } e \text{ and } X}{\text{Count of samples consistent with } e}$$
  - **Advantages:** Simple to implement for small networks.
  - **Disadvantages:** Inefficient for large networks or when evidence is rare, leading to many rejected samples.
- **Likelihood Weighting:**
  - Samples only non-evidence variables while fixing evidence variables.
  - Assigns a weight w to each sample based on the likelihood of the evidence given the sampled values.
  - **Weight Calculation:**
  $$w = \prod_{E_i \in e} P(E_i|\text{parents}(E_i))$$
  where E_i are evidence variables.
  - **Estimated Probability:**
  $$P(X|e) \approx \frac{\sum_{\text{samples}} w \cdot \mathbb{I}(X)}{\sum_{\text{samples}} w}$$
  where $\mathbb{I}(X)$ is an indicator function that is 1 if X is true in the sample and 0 otherwise.
  - **Advantages:** More efficient than rejection sampling for networks with rare evidence.
  - **Disadvantages:** Accuracy degrades if evidence variables strongly constrain the network.
- **Gibbs Sampling:**
  - Starts with an initial assignment of all variables.
  - Iteratively samples each variable conditioned on the current values of all other variables:
  $$P(X_i|\mathbf{X}_{-i}, e) \propto P(X_i|\text{parents}(X_i)) \prod_{Y_j \in \text{children}(X_i)} P(Y_j|\text{parents}(Y_j))$$
  where $\mathbf{X}_{-i}$ denotes all variables except X_i.
  - After a **burn-in period**, the samples approximate the target distribution.
  - **Advantages:** Effective for complex networks with many variables and observed evidence.
  - **Disadvantages:** Convergence can be slow when variables are highly interdependent.

7 Machine Learning

> *"Machine learning is the science of getting computers to act without being explicitly programmed."*
>
> – Tom M. Mitchell

7.1 K-Nearest Neighbors (KNN)

Definition

K-Nearest Neighbors is an intuitive machine learning method used for both classification and regression. It works by looking at the closest points around a new data point and making decisions based on them.

Key Characteristics

- Makes no assumptions about the data distribution.
- Does not build a model during training but stores the training dataset for later computations.
- Uses specific instances from the training data to make predictions for new data points.

How it Works

Given a query point (the point for which we want to predict the class/label):

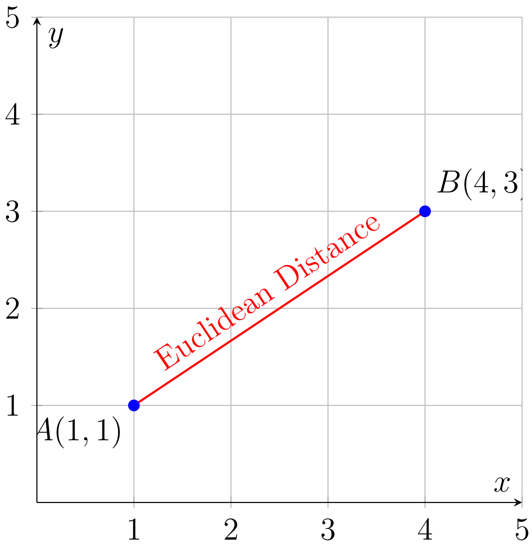
1. Calculate the distance between the query point and all points in the training dataset.
2. Select the K nearest points (neighbors) based on the distance metric.
3. For classification, assign the class with the majority vote among the neighbors. For regression, average the values of the nearest neighbors.

Distance Metrics

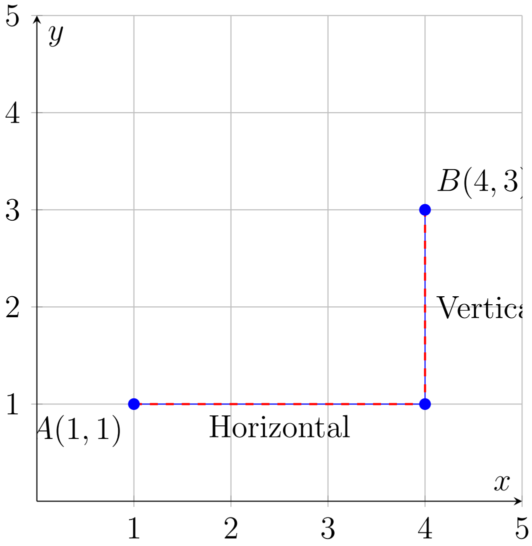
- **Euclidean Distance:**

  $$d(x_1, x_2) = \sqrt{\sum (x_1 - x_2)^2}$$

  Example: Euclidean distance is the straight-line distance between two points in a Cartesian plane.



- **Manhattan Distance:**
$$d(x_1, x_2) = \sum |x_1 - x_2|$$
Example: Manhattan distance measures the distance along grid-like paths (like city blocks).

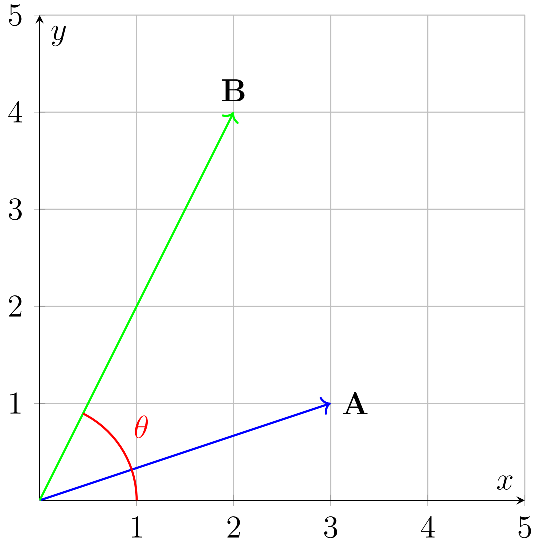


- **Minkowski Distance:**
$$d(x_1, x_2) = \left(\sum |x_1 - x_2|^p\right)^{1/p}$$
Example: A generalization of Euclidean and Manhattan distances, controlled by parameter p. For $p = 1$, it becomes Manhattan distance; for $p = 2$, it becomes Euclidean distance.

(Graph omitted; similar to Euclidean and Manhattan distance, depending on p.)

- **Cosine Similarity:**
$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

Example: Cosine similarity measures the angle between two vectors, focusing on their direction rather than magnitude.



Choosing K

- A small K (e.g., 1) makes KNN sensitive to noise, leading to overfitting.
- A large K (e.g., too large) can cause underfitting, making the algorithm less sensitive to local structure.
- **Common Practice:** Try different values of K and choose the one that performs best using cross-validation.

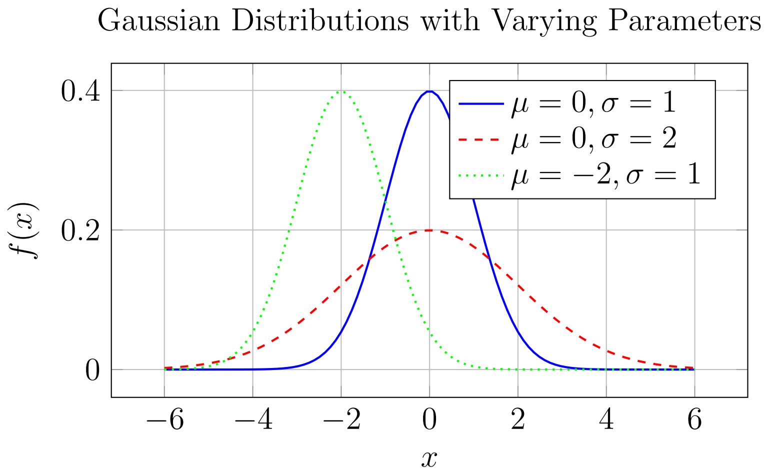
Underfitting, Overfitting, and Cross-Validation

- **Underfitting:** Occurs when the model is too simple to capture patterns in the data (e.g., very large K).
- **Overfitting:** Happens when the model is too specific to the training data (e.g., very small K).
- **Cross-Validation:** Helps select the optimal value of K by splitting the training data into subsets and evaluating model performance.

K-NN Advantages

- **Simplicity:** Easy to understand and implement.
- **No Training Phase:** Useful for scenarios where the training set updates frequently.
- **Flexibility:** Applicable to both classification and regression tasks.

7.2 The Gaussian Distribution



$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- **Mean** (μ)**:** Represents the expected value or average of the data points.
- **Standard Deviation** (σ)**:** Measures the spread of the data around the mean.
- **Symmetry:** The Gaussian distribution is symmetric, often assumed in linear models.
- **68-95-99.7 Rule:** Describes the proportion of data within one, two, and three standard deviations from the mean.

7.3 Central Limit Theorem (CLT)

Definition: The CLT states that, regardless of the original distribution, the sampling distribution of the sample mean will approach a normal distribution as the sample size becomes large.

- **Model Assumptions:** Many machine learning models assume normally distributed errors or residuals.
- **Improved Estimation:** The sample mean approximates the population mean for large samples.
- **Standardized Data:** CLT supports normalizing data for algorithms that perform better with Gaussian inputs.

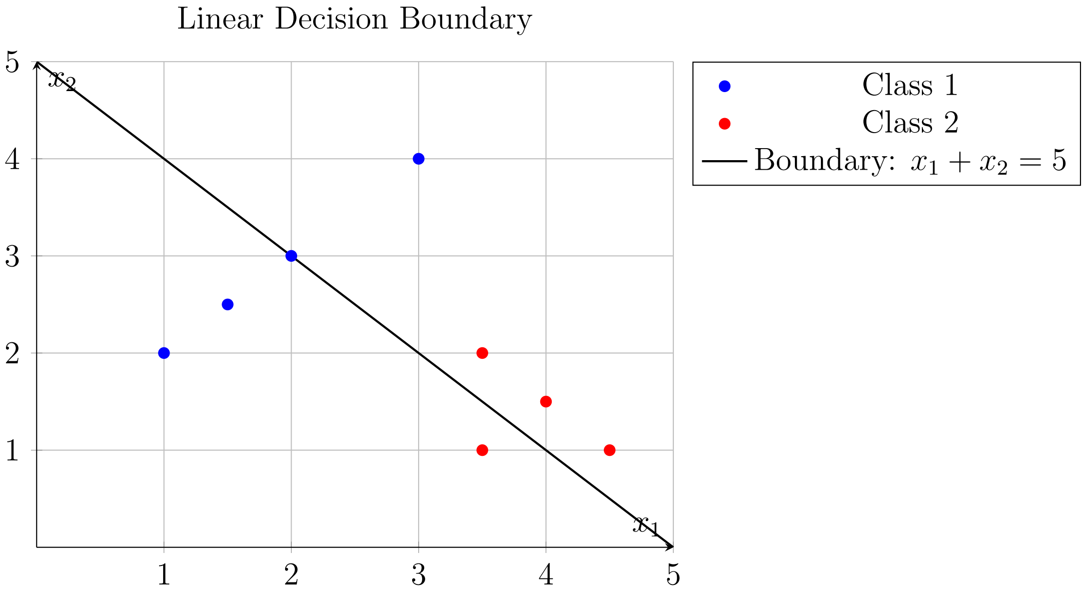
7.4 Decision Boundaries

Definition: A decision boundary is a surface that separates different classes in a classification problem.

- **Linear:** Straight lines or planes, e.g., logistic regression.
- **Non-linear:** Curved or complex shapes, e.g., decision trees or neural networks.

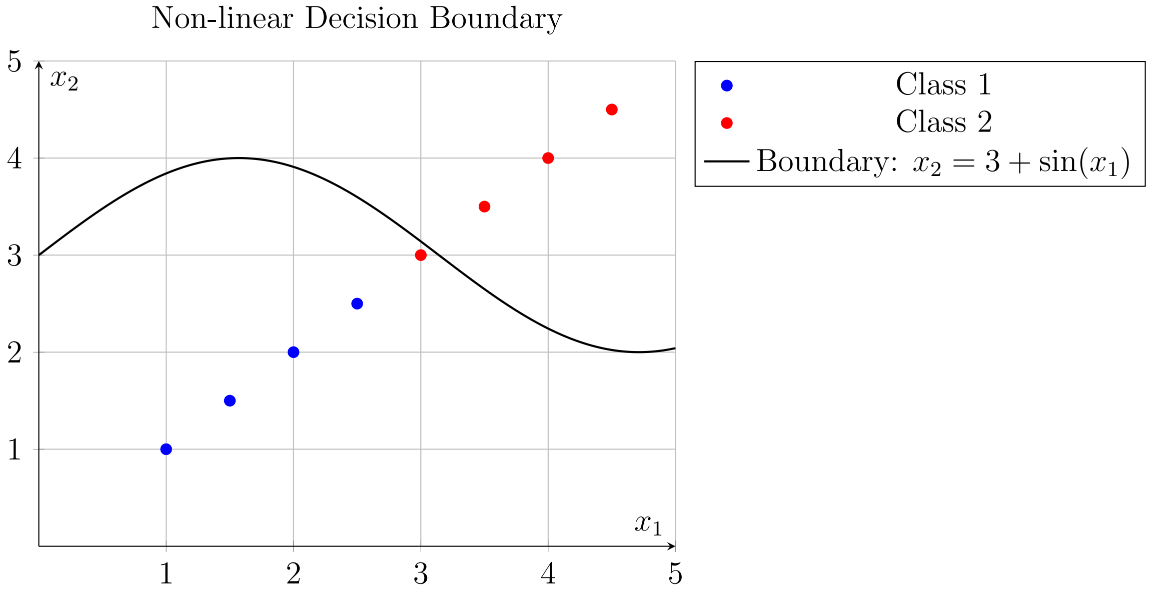
Linear Decision Boundary

A linear decision boundary separates classes with a straight line or plane.



7.5 Non-linear Decision Boundary

A non-linear decision boundary separates classes with a curve or complex shape.



7.6 Bayes Classifier

$$P(C_k|x) = \frac{P(x|C_k) \cdot P(C_k)}{P(x)}$$

- **Posterior Probability $(P(C_k|x))$:** Probability of class C_k given data x.
- **Likelihood $(P(x|C_k))$:** Probability of data x given class C_k.

- **Prior** $(P(C_k))$**:** Probability of class C_k without considering data.
- **Marginal Likelihood** $(P(x))$**:** Probability of the data x across all classes.

7.7 Maximum Likelihood Estimation (MLE)

$$\hat{\theta}_{ML} = \arg\max_{\theta} L(\theta|X) = \arg\max_{\theta} P(X|\theta)$$

- $\hat{\theta}_{ML}$: The parameter that maximizes the likelihood function.
- $L(\theta|X)$: Likelihood of the data given parameter θ.
- $P(X|\theta)$: Probability of observing the data given θ.

7.8 When is Maximum Likelihood Estimation Used?

MLE is widely used in statistical modeling and machine learning for parameter estimation. It is particularly effective when:

- The form of the probability distribution is known (e.g., Gaussian, Poisson, etc.).
- The goal is to find the parameter values that make the observed data most probable.
- Data is assumed to be generated from an underlying probabilistic model.

7.9 Applications of MLE

- **Linear Regression:** Estimating coefficients when assuming normally distributed errors.
- **Logistic Regression:** Determining weights by maximizing the likelihood of binary classifications.
- **Gaussian Mixture Models:** Estimating the parameters of component distributions in a mixture model.
- **Hidden Markov Models:** Learning transition and emission probabilities using observed data.
- **Neural Networks:** Optimizing weights via maximum likelihood principles (often approximated using stochastic gradient descent).

7.10 Random Forests

Random Forest is an ensemble learning technique that builds multiple decision trees during training and combines their outputs for prediction. It reduces overfitting and improves accuracy.

7.11 Key Features

- **Bootstrap Aggregation (Bagging):** Each tree is trained on a random subset of the data (with replacement).
- **Random Feature Selection:** At each split, only a random subset of features is considered.
- **Final Prediction:**
  - **Classification:** The final output is the **mode** of the predictions from all trees:

    $$\hat{y} = \text{mode}(h_1(x), h_2(x), \ldots, h_T(x))$$

  - **Regression:** The final output is the **mean** of the predictions from all trees:

    $$\hat{y} = \frac{1}{T}\sum_{t=1}^{T} h_t(x)$$

    where T is the total number of trees.
- **OOB (Out-of-Bag) Error:** The error is estimated using data not included in the bootstrap sample.

7.12 Advantages

- Reduces overfitting compared to a single decision tree.
- Works well for both classification and regression tasks.
- Robust to noise and missing data.

7.13 Equations for Random Forest

- **Gini Impurity:** Used to evaluate splits in decision trees:

  $$Gini = 1 - \sum_{i=1}^{C} p_i^2$$

  where p_i is the proportion of instances in class i, and C is the number of classes.
- **Weighted Gini for Splits:**

  $$Gini_{\text{split}} = \frac{n_{\text{left}}}{n_{\text{total}}} Gini_{\text{left}} + \frac{n_{\text{right}}}{n_{\text{total}}} Gini_{\text{right}}$$

- **Entropy (for information gain):**

  $$Entropy = -\sum_{i=1}^{C} p_i \log_2(p_i)$$

7.14 Neural Networks

A Neural Network is a computational model inspired by the human brain. It consists of layers of interconnected nodes (neurons) that process input data and produce predictions.

Perceptron

Definition: The Perceptron is the simplest type of neural network, used for binary classification tasks.

- **Single-Layer Perceptron:**
  - Performs a weighted sum of inputs and applies an activation function (e.g., step function).
  - Can only classify linearly separable data.
- **Multilayer Perceptron (MLP):**
  - Adds hidden layers to learn more complex patterns.
  - Uses non-linear activation functions (e.g., ReLU, sigmoid, tanh).
  - Capable of solving non-linearly separable problems.

7.15 Backpropagation

Definition: Backpropagation is the algorithm used to train neural networks by minimizing the error between predicted and actual values.

- **Steps:**
  1. Perform forward propagation to compute the output.
  2. Calculate the loss (e.g., mean squared error).
  3. Propagate the error backward to adjust weights using the gradient of the loss function.
- **Optimization:** Common methods include gradient descent, stochastic gradient descent (SGD), and Adam optimizer.

7.16 Equations

- **Weighted Sum:**

  $$z = \sum_{i=1}^{n} w_i x_i + b$$

  where w_i are weights, x_i are inputs, and b is the bias.

- **Activation Function:**

  $$a = f(z)$$

  Common choices for $f(z)$:

  - **Sigmoid**: $\sigma(z) = \frac{1}{1+e^{-z}}$

  - **ReLU**: $f(z) = \max(0, z)$
  - **Tanh**: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

- **Loss Function:**

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

  where y_i is the actual value and $\hat{y}_i$ is the predicted value.

- **Weight Update Rule:**

$$w_i \leftarrow w_i - \eta \frac{\partial \mathcal{L}}{\partial w_i}$$

  where η is the learning rate.

7.17 Clustering

Clustering is an unsupervised learning technique used to group data points into clusters based on their similarity.

7.18 K-means Clustering

- Iteratively assigns data to clusters and updates centroids.
- Requires specifying the number of clusters, k, beforehand.
- Converges when cluster assignments no longer change or centroids stabilize.
- Sensitive to initial centroid placement and may converge to a local optimum.
- Suitable for spherical clusters with similar sizes.

7.19 Expectation-Maximization (EM)

- Soft clustering by estimating probabilities and maximizing likelihood.
- Alternates between the Expectation step (E-step), which calculates the probability of data points belonging to each cluster, and the Maximization step (M-step), which updates cluster parameters to maximize the likelihood.
- Suitable for clustering data that follows a probabilistic distribution, such as Gaussian Mixture Models (GMMs).
- Provides flexibility in cluster shapes compared to K-means.
- Computationally more intensive than K-means.

8 Pattern Recognition Through Time

> *"Pattern recognition is the ability to detect order in chaos by finding structures that persist through time."*
>
> – Christopher Bishop

Warping Time

Time warping is a powerful concept in time-series analysis and pattern recognition, where temporal misalignments between sequences are corrected to enable meaningful comparisons.

Temporal Patterns

Temporal patterns are sequences of events or data points occurring over time, exhibiting regularity or structure. Examples include:

- Variations in pitch, rhythm, heartbeats, stock prices, walking patterns, etc.
- Complex interactions such as sensor data from multiple systems or cyclic economic indicators.

Temporal patterns often contain variations in speed, amplitude, or noise, making direct comparisons challenging.

8.1 Dynamic Time Warping (DTW)

Dynamic Time Warping (DTW) is a technique for aligning two temporal sequences by "stretching" or "compressing" time to minimize their differences. It finds the optimal alignment between two sequences by constructing a warping path in a cost matrix.

Applications of DTW

- **Speech Recognition**: Aligning spoken words for comparison despite variations in speaking speed.
- **Time-Series Analysis**: Aligning stock market trends or sensor data.

DTW Algorithm Steps

1. **Cost Matrix Construction**: Compute the pairwise distance between elements of two sequences,

   $$X = [x_1, x_2, \ldots, x_n] \quad \text{and} \quad Y = [y_1, y_2, \ldots, y_m],$$

   to form a cost matrix D, where

   $$D(i, j) = \|x_i - y_j\|.$$

2. **Dynamic Programming Recurrence**: Compute the cumulative cost $C(i, j)$ for aligning x_i with y_j:

$$C(i,j) = D(i,j) + \min \begin{cases} C(i-1, j), \\ C(i, j-1), \\ C(i-1, j-1) \end{cases}$$

   where $C(0, 0) = 0$, and boundary conditions are handled appropriately.

3. **Warping Path Extraction**: Backtrack from $C(n, m)$ to $C(1, 1)$ to find the optimal warping path,

$$\mathcal{W} = [(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)],$$

   which minimizes the total alignment cost.

Visualizing DTW Alignment A cost matrix D can be visualized with the optimal warping path $\mathcal{W}$ as a line traversing from $(1, 1)$ to (n, m), showing how the sequences align.

Example Alignment: Cost Matrix with Optimal Warping Path $\mathcal{W}$

$$\begin{bmatrix} C(1,1) & C(1,2) & C(1,3) & \cdots & C(1,m) \\ C(2,1) & C(2,2) & C(2,3) & \cdots & C(2,m) \\ C(3,1) & C(3,2) & C(3,3) & \cdots & C(3,m) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C(n,1) & C(n,2) & C(n,3) & \cdots & C(n,m) \end{bmatrix}$$

$$\mathcal{W} = \{(1,1), (2,2), (3,3), \ldots, (n,m)\}$$

Key: Cells highlighted in gray represent the optimal warping path.

This approach ensures robust comparison, even under non-linear temporal distortions, enhancing the utility of time-series analysis.

8.2 Sakoe-Chiba Bounds

Sakoe-Chiba bounds constrain the warping path in Dynamic Time Warping (DTW) to a fixed-width band around the diagonal of the cost matrix. This serves two main purposes:

- **Reduce Computational Cost:** By limiting the number of cells considered during alignment.
- **Prevent Unrealistic Alignments:** By restricting excessive stretching or compressing of the sequences.

Concept

The Sakoe-Chiba band defines a fixed-width region of the DTW cost matrix within which the warping path is allowed to traverse. For a band of width w, the warping path is constrained to satisfy:

$$|i - j| \leq w, \quad \forall (i, j) \in \mathcal{W},$$

where $\mathcal{W}$ represents the warping path, and i and j are indices of the two sequences being aligned.

Cost Matrix with Sakoe-Chiba Band:

$$\begin{bmatrix} C(1,1) & C(1,2) & \cdot & \cdot & \cdots \\ C(2,1) & C(2,2) & C(2,3) & \cdot & \cdots \\ \cdot & C(3,2) & C(3,3) & C(3,4) & \cdots \\ \vdots & \cdot & C(4,3) & C(4,4) & C(4,5) \\ \cdots & \vdots & \cdots & C(n-1,m-2) & C(n,m) \end{bmatrix}$$

Cells outside the band (unshaded) are ignored during the computation, drastically reducing the number of operations.

Advantages

- **Efficiency:** By constraining the warping path, the Sakoe-Chiba band reduces the computational complexity from $O(n \times m)$ to approximately $O(w \times n)$, where w is the width of the band.
- **Improved Accuracy:** Restricts excessive warping that might lead to unnatural alignments, ensuring that only meaningful temporal distortions are considered.

Width Parameter

The width of the Sakoe-Chiba band, w, controls the trade-off between alignment flexibility and computational efficiency:

- **Small w:** Faster computation but less flexibility, potentially missing valid alignments.
- **Large w:** More flexibility but increased computation, approaching unconstrained DTW.

The choice of w depends on the expected variability in the sequences and the application context.

Visualization

A visualization of the warping path constrained by the Sakoe-Chiba band within the cost matrix is shown below:

Warping Path with Sakoe-Chiba Band

	y_1	y_2	y_3	y_4
x_1	$C(1,1)$	$C(1,2)$	$\cdot$	$\cdot$
x_2	$C(2,1)$	$C(2,2)$	$C(2,3)$	$\cdot$
x_3	$\cdot$	$C(3,2)$	$C(3,3)$	$C(3,4)$
x_4	$\cdot$	$\cdot$	$C(4,3)$	$C(4,4)$

Cells highlighted in red indicate the allowable region of the warping path constrained by the band width w.

Equation Summary

For two sequences $X = [x_1, x_2, \ldots, x_n]$ and $Y = [y_1, y_2, \ldots, y_m]$, with Sakoe-Chiba bounds:

$$C(i,j) = \begin{cases} D(i,j) + \min \begin{cases} C(i-1,j) \\ C(i,j-1) \\ C(i-1,j-1) \end{cases} & , \quad \text{if } |i-j| \leq w, \\ \infty, & \quad \text{otherwise.} \end{cases}$$

This ensures computations are restricted to the defined band.

8.3 Viterbi Algorithm

Purpose

Finds the most likely sequence of hidden states given observed events in HMMs.

Approach

- Uses dynamic programming to maximize the probability of reaching each state in sequence.
- Visualized using a **Trellis Diagram**.

Formula

$$\delta_t(j) = \max_i \left[\delta_{t-1}(i) \times a_{ij}\right] \times b_j(o_t)$$

Where:

- $\delta_t(j)$: Maximum probability of any path reaching state j at time t given observations up to t.

- a_{ij}: Transition probability from state i to j.
- $b_j(o_t)$: Emission probability of observing o_t in state j.

Matrix Representations

Transition Matrix (A):

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

1. a_{11}: Probability of staying in S_1 (Sunny to Sunny).
2. a_{12}: Probability of transitioning from S_1 to S_2 (Sunny to Rainy).
3. a_{21}: Probability of transitioning from S_2 to S_1 (Rainy to Sunny).
4. a_{22}: Probability of staying in S_2 (Rainy to Rainy).

Emission Matrix (B):

$$B = \begin{bmatrix} b_1(O_1) & b_1(O_2) \\ b_2(O_1) & b_2(O_2) \end{bmatrix}$$

1. $b_1(O_1)$: Probability of observing O_1 (Happy) from S_1 (Sunny).
2. $b_1(O_2)$: Probability of observing O_2 (Sad) from S_1 (Sunny).
3. $b_2(O_1)$: Probability of observing O_1 (Happy) from S_2 (Rainy).
4. $b_2(O_2)$: Probability of observing O_2 (Sad) from S_2 (Rainy).

Steps of the Algorithm

1. **Initialization:** Compute the initial probabilities for each state:

   $$\delta_1(j) = \pi_j \times b_j(o_1)$$

   where π_j is the initial probability of state j, and $b_j(o_1)$ is the probability of the first observation in state j.

2. **Recursion:** For each subsequent observation t, compute:

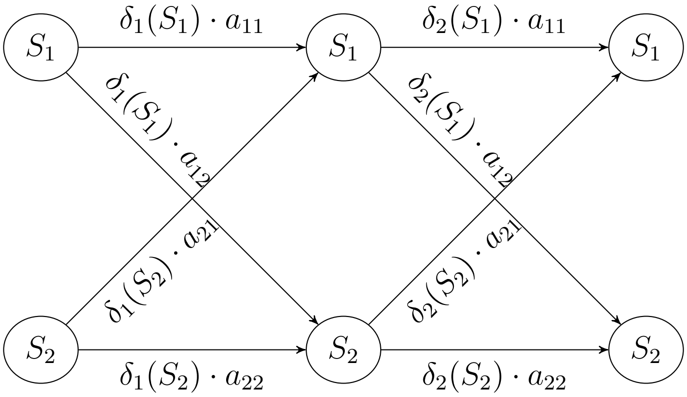
   $$\delta_t(j) = \max_i \left[\delta_{t-1}(i) \times a_{ij}\right] \times b_j(o_t)$$

3. **Termination:** Find the most probable final state:

   $$P_{\max} = \max_j \delta_T(j)$$

4. **Backtracking:** Trace back through the stored states to reconstruct the most probable path.

Trellis Diagram


S_1
S_2
$\delta_1(S_1) \cdot a_{11}$
$\delta_2(S_1) \cdot a_{11}$
$\delta_1(S_1) \cdot a_{12}$
$\delta_2(S_1) \cdot a_{12}$
$\delta_1(S_2) \cdot a_{21}$
$\delta_2(S_2) \cdot a_{21}$
$\delta_1(S_2) \cdot a_{22}$
$\delta_2(S_2) \cdot a_{22}$

8.4 Baum-Welch Algorithm

Purpose

Trains Hidden Markov Models (HMMs) by estimating the parameters of the model (transition probabilities A, emission probabilities B, and initial state probabilities π) to maximize the likelihood of observed data.

Type

An example of the **Expectation-Maximization (EM)** algorithm, which iteratively refines parameters to maximize the likelihood of observed data.

Steps

1. **E-step (Expectation):** Calculates the expected probabilities of being in a given hidden state at a particular time, given the current model parameters and observations.

   - **Forward probability ($\alpha_t(i)$):** Probability of reaching state i at time t and observing the sequence up to t:

$$\alpha_t(i) = \sum_{j=1}^{N} \alpha_{t-1}(j) \cdot a_{ji} \cdot b_i(o_t)$$

   - **Backward probability ($\beta_t(i)$):** Probability of observing the sequence from time $t+1$ to the end, given the state at time t:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} \cdot b_j(o_{t+1}) \cdot \beta_{t+1}(j)$$

   - **State occupancy probability ($\gamma_t(i)$):** Probability of being in state i at time t:

$$\gamma_t(i) = \frac{\alpha_t(i) \cdot \beta_t(i)}{\sum_{k=1}^{N} \alpha_t(k) \cdot \beta_t(k)}$$

- **State transition probability ($\xi_t(i,j)$):** Probability of transitioning from state i to state j at time t:

$$\xi_t(i,j) = \frac{\alpha_t(i) \cdot a_{ij} \cdot b_j(o_{t+1}) \cdot \beta_{t+1}(j)}{\sum_{k=1}^{N} \sum_{l=1}^{N} \alpha_t(k) \cdot a_{kl} \cdot b_l(o_{t+1}) \cdot \beta_{t+1}(l)}$$

2. **M-step (Maximization):** Updates the model parameters using the probabilities computed in the E-step:

   - **Transition probabilities:**

$$a_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

   - **Emission probabilities:**

$$b_j(o_k) = \frac{\sum_{t=1}^{T} \gamma_t(j) \cdot \nVdash(o_t = o_k)}{\sum_{t=1}^{T} \gamma_t(j)}$$

   where $\nVdash(o_t = o_k)$ is an indicator function that is 1 if $o_t = o_k$, and 0 otherwise.

   - **Initial state probabilities:**

$$\pi_i = \gamma_1(i)$$

3. **Iteration:** Repeat the E and M steps until the parameters converge (i.e., the change in the log-likelihood of the observations stabilizes).

Applications

Widely used in:

- **Speech recognition:** Training acoustic models to predict phonemes from audio.
- **Bioinformatics:** Inferring gene structures or protein sequences.
- **Natural language processing:** Part-of-speech tagging, machine translation, etc.

8.5 Stochastic Beam Search

Purpose

Generates diverse sequences in NLP tasks by introducing randomness into beam search.

Mechanism

- Expands a fixed number of paths by stochastically sampling from top candidates at each step.
- Balances probability with randomness to retain diverse paths.

Equations

In stochastic beam search:

- At each step t, compute the probabilities of possible next tokens:

  $$P(y_t|y_{<t}, x) = \frac{\exp(s(y_t|y_{<t}, x)/\tau)}{\sum_{y' \in \mathcal{Y}_t} \exp(s(y'|y_{<t}, x)/\tau)}$$

  where:

  - $s(y_t|y_{<t}, x)$: Score of token y_t given context $y_{<t}$ and input x.
  - τ: Temperature parameter to control randomness.
  - $\mathcal{Y}_t$: Set of possible tokens at time t.

- Stochastically sample k candidates based on probabilities $P(y_t|y_{<t}, x)$.

Pros and Cons

- **Pros:**
  - Increases output variety, useful for tasks like machine translation and text generation.
  - Produces diverse yet coherent outputs by balancing probability and randomness.
- **Cons:**
  - May introduce suboptimal paths if randomness dominates.
  - Requires careful tuning of the temperature τ and beam size k.

9 Hidden Markov Models (HMMs)

> *"Hidden Markov Models provide a simple and effective framework for modeling time-series data where the system being modeled is assumed to follow a Markov process with hidden states."*
>
> – Lawrence R. Rabiner

9.1 Markov Chain

A model for sequences where each event depends only on the previous one.

Key Components:

- **States:** Possible situations (e.g., weather: cloudy, rainy, sunny).
- **Transitions:** Probabilities of moving from one state to another.
- **Transition Matrix:** Table of transition probabilities.

9.2 Definitions

A Hidden Markov Model (HMM) is a statistical model where the system is assumed to follow a Markov process with hidden (unobservable) states that produce observable outputs. It is defined by:

- A set of **hidden states** $\{S_1, S_2, \ldots, S_N\}$.
- A set of **observations** $\{O_1, O_2, \ldots, O_M\}$ produced by the hidden states.
- A **transition probability matrix** $A = [a_{ij}]$, where a_{ij} is the probability of transitioning from state S_i to S_j.
- An **emission probability matrix** $B = [b_j(O_k)]$, where $b_j(O_k)$ is the probability of observing O_k given the system is in state S_j.
- An **initial state distribution** $\pi = [\pi_i]$, where π_i is the probability of starting in state S_i.

Example: Consider a weather system:

- **Hidden states:** Weather conditions (e.g., Sunny, Rainy).
- **Observations:** A friend's mood (e.g., Happy, Sad).
- **Goal:** Infer the hidden sequence of weather states based on the sequence of observed moods.

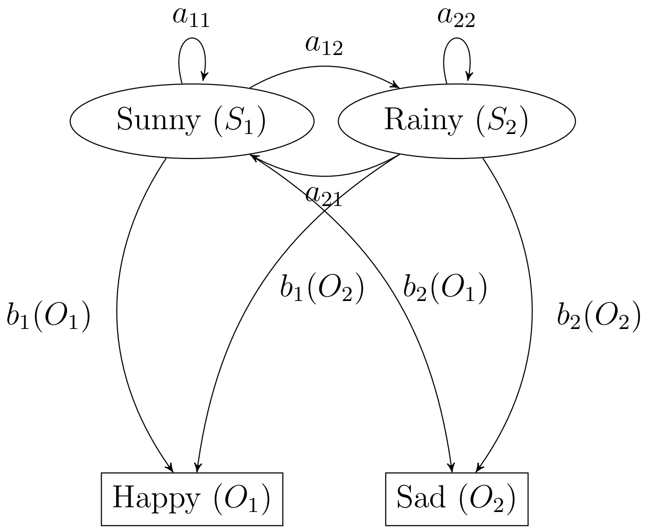
Inference Tasks in HMMs:

1. **Likelihood Estimation:** Calculate the probability of a sequence of observations.
2. **Decoding:** Determine the most likely sequence of hidden states (e.g., Viterbi algorithm).
3. **Learning:** Estimate the model parameters (A, B, π) given a set of observations.

9.3 HMM Diagram

Below is a visualization of a Hidden Markov Model using a weather example:

HMM Diagram: Hidden States and Observations Below is a visualization of a Hidden Markov Model using a weather example:



Explanation of the Diagram:

1. **hidden states** (Sunny, Rainy) are shown as ellipses.
2. **observations** (Happy, Sad) are shown as rectangles.
3. **Transition probabilities** a_{ij} define the likelihood of moving between hidden states.
4. **Emission probabilities** $b_j(O_k)$ define the likelihood of observing a specific observation from a given state.

Mathematical Representation: The joint probability of a sequence of observations $O = [O_1, O_2, \ldots, O_T]$ and hidden states $S = [S_1, S_2, \ldots, S_T]$ is given by:

$$P(O, S) = \pi_{S_1} \prod_{t=1}^{T-1} a_{S_t S_{t+1}} \prod_{t=1}^{T} b_{S_t}(O_t),$$

where:

1. π_{S_1} is the initial probability of starting in state S_1.
2. $a_{S_t S_{t+1}}$ is the transition probability from state S_t to S_{t+1}.
3. $b_{S_t}(O_t)$ is the emission probability of observing O_t from state S_t.

10 Logic and Planning

> *”Intelligence is the ability to solve problems using knowledge, reasoning, and planning.”*
>
> – Stuart Russell Peter Norvig

10.1 Propositional Logic

Propositional logic is a formal system in logic used to represent statements and their relationships using propositions and logical connectives. Below are the key components and rules of propositional logic.

10.2 Key Components

- **Proposition**: a statement that is either **true** (T) or **false** (F). Example: “It is raining.”
- **Logical Connectives**:
  - Negation ($\neg p$): Logical NOT. True if p is false.
  - Conjunction ($p \wedge q$): Logical AND. True if both p and q are true.
  - Disjunction ($p \vee q$): Logical OR. True if at least one of p or q is true.
  - Implication ($p \rightarrow q$): Logical IF-THEN. True unless p is true and q is false.
  - Biconditional ($p \leftrightarrow q$): Logical IF AND ONLY IF. True if p and q have the same truth value.

10.3 Truth Tables

Below is the truth table for the propositions P and Q, along with the logical operations: $\neg P$ (NOT P), $P \wedge Q$ (AND), $P \vee Q$ (OR), $P \rightarrow Q$ (implies), and $P \leftrightarrow Q$ (biconditional).

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

10.4 Examples

- O: 5 is an odd number.
- P: Paris is the capital of France.
- $(O \rightarrow P)$: True or False?

    - **Answer:** 5 is an odd number, so $O =$ True. Paris is the capital of France, so $P =$ True. Therefore, $T \rightarrow T = T$.

- E: 5 is an even number.

- M: Moscow is the capital of France.

- $(E \rightarrow M)$: True or False?

    - **Answer:** 5 is NOT an even number, so $E =$ False. Moscow is NOT the capital of France, so $M =$ False. Therefore, $F \rightarrow F = T$.

Truth Table

P	Q	$P \wedge (P \rightarrow Q)$	$\neg(\neg P \vee \neg Q)$	$(P \wedge (P \rightarrow Q)) \iff (\neg(\neg P \vee \neg Q))$
F	F	F	F	T
F	T	F	F	T
T	F	F	F	T
T	T	T	T	T

10.5 Limitations of Propositional Logic

Propositional logic, while foundational and useful in many contexts, has several limitations:

- **Lack of Quantifiers:** Propositional logic does not support quantifiers such as $\forall$ (for all) or $\exists$ (there exists), which are essential for reasoning about collections or statements involving variables.

- **No Internal Structure of Statements:** In propositional logic, statements are treated as atomic units without any internal structure. For example, the statement ”John is tall” cannot be analyzed further to understand its components (e.g., subject, predicate).

- **Inability to Represent Relationships:** Propositional logic cannot express relationships between objects. For example, ”John is taller than Mary” cannot be expressed using propositional logic alone.

- **Limited Expressiveness:** Complex mathematical and real-world reasoning often require richer frameworks such as first-order logic (predicate logic) or higher-order logic, which propositional logic cannot accommodate.

- **Scalability Issues:** As the number of propositions increases, the truth table approach for evaluating logical expressions becomes computationally infeasible due to exponential growth in rows.

- **No Handling of Uncertainty:** Propositional logic is strictly binary (true or false). It cannot model uncertainty or degrees of truth, which are better handled by probabilistic logic or fuzzy logic.

Despite these limitations, propositional logic serves as a critical foundation for more advanced forms of logic and computational reasoning. Its simplicity and clarity make it a useful tool for introducing logical concepts and reasoning patterns.

10.6 Comparison of Propositional Logic, First-Order Logic, and Probability Theory

Language	**Ontological Commitment (What exists in the world)**	**Epistemological Commitment (What an agent believes about facts)**
Propositional logic	Facts	True/False/Unknown
First-order logic	Facts, objects, relations	True/False/Unknown
Temporal logic	Facts, objects, relations, times	True/False/Unknown
Probability theory	Facts	Degree of belief $\in [0, 1]$
Fuzzy logic	Facts with degree of truth $\in [0, 1]$	Known interval value

10.7 Example of a Model in First-Order Logic

A model in First-Order Logic consists of a domain of discourse, objects, predicates, and functions. Below is an example of a simple model:

- **Domain:** $\{Alice, Bob, Charlie\}$ (people in the model).
- **Predicates:**
  - $Likes(x, y)$: Represents that person x likes person y.
  - $Friend(x, y)$: Represents that person x is a friend of person y.
- **Functions:**
  - $Parent(x)$: Represents the parent of person x.
- **Statements:**
  - $\forall x\, (Likes(x, Alice) \rightarrow Friend(x, Alice))$: "Everyone who likes Alice is her friend."

- $\exists x\,(Likes(Bob, x) \wedge Friend(Bob, x))$: "There exists someone whom Bob likes and is also Bob's friend."
- $Likes(Alice, Bob)$: "Alice likes Bob."
- $Parent(Charlie) = Alice$: "Alice is the parent of Charlie."

- **Interpretation:**
  - $Likes(Alice, Bob) =$ True.
  - $Likes(Bob, Alice) =$ False.
  - $Friend(Alice, Bob) =$ True.
  - $Parent(Charlie) = Alice$.

Explanation: This model represents a small social network where relationships between individuals are described using predicates like *Likes* and *Friend*. Functions like *Parent* capture more complex relationships. Quantified statements allow reasoning about all or some members of the domain.

10.8 Analysis of Logical Statements

1. $\exists x, y\ x = y$ **Validity: Valid**
   This statement is valid because we can always assign $x = y$ to be any element in a non-empty domain, satisfying the condition $x = y$.

2. $(\exists x\ x = x) \implies (\forall y \exists z\ y = z)$ **Validity: Valid**
   The antecedent $\exists x\ x = x$ is always true since every object in any domain is equal to itself. The consequent $\forall y \exists z\ y = z$ is also always true because we can take $z = y$ for any y in the domain, making the implication valid.

3. $\forall x\ P(x) \vee \neg P(x)$ **Validity: Valid**
   This statement is valid as it is an instance of the law of the excluded middle in classical logic, stating that for any proposition $P(x)$, either $P(x)$ or $\neg P(x)$ must hold.

4. $\exists x\ P(x)$ **Satisfiability: Satisfiable, but not valid**
   This statement is satisfiable if there exists at least one x in the domain for which $P(x)$ is true. However, it is not valid because it depends on the actual domain and predicate $P(x)$—if the domain is empty or $P(x)$ is false for all x, it would be unsatisfiable.

5. $\exists x, y\ \text{Job}(\text{Sam}, x) \wedge \text{Job}(\text{Sam}, y) \wedge \neg(x = y)$ **Correctness: Correct**
   This statement correctly represents that Sam has two distinct jobs. It asserts that there exist two distinct entities x and y for which the predicate $\text{Job}(\text{Sam}, \cdot)$ holds, and $x \neq y$ ensures that they are different.

6. $\forall x, s\ \text{Member}(x, \text{Add}(x, s))$ **Correctness: Correct**
   This statement correctly represents that x is a member of the set resulting from adding x to the set s. It explicitly states that for any x and s, x belongs to $\text{Add}(x, s)$.

7. $\forall x, s\ \text{Member}(x, s) \implies (\forall y\ \text{Member}(x, \text{Add}(y, s)))$ **Correctness: Incorrect** This statement does not correctly represent set membership. It implies that if x is a member of s, then x must also be a member of the set resulting from adding any element y to s, which is not necessarily true in standard set theory.

10.9 Stochastic, Multi-Agent, Partial Observability in Logic

1. **Stochastic Systems:** A stochastic system is one in which outcomes are determined by probabilistic transitions. Formally, this can be expressed as:

$$\forall s, a, s'\ P(s' \mid s, a) \geq 0 \wedge \sum_{s'} P(s' \mid s, a) = 1$$

Here, $P(s' \mid s, a)$ represents the probability of transitioning to state s' from state s after action a. This captures the uncertainty inherent in the system.

2. **Multi-Agent Systems:** Multi-agent systems involve multiple decision-making entities. Each agent i has its own actions a_i and possibly its own utility or objective function $U_i(s)$. Interactions can be expressed as:

$$\forall i, s, a_i\ U_i(s) = f(s, a_1, \ldots, a_n)$$

where f describes the joint effects of the actions of all agents. Coordination, competition, or collaboration among agents can be modeled depending on f and the domain.

3. **Partial Observability:** In partially observable environments, agents do not have full access to the system state s but instead receive observations o drawn from an observation function:

$$\forall s, o, a\ P(o \mid s, a) \geq 0 \wedge \sum_{o} P(o \mid s, a) = 1$$

The agent's belief about the state can be updated using a Bayesian filter:

$$b'(s') = \eta \cdot P(o \mid s', a) \sum_{s} P(s' \mid s, a) b(s)$$

where $b(s)$ is the belief over the state, $b'(s')$ is the updated belief, and η is a normalizing constant.

10.10 Forward and Backward Chaining

Forward Chaining

Forward chaining is a data-driven approach that starts with known facts and applies inference rules to extract more data until a goal is reached.

Algorithm:

1. Initialize the working memory with known facts.
2. While the goal is not reached:
    - Identify rules where all premises are true based on the working memory.
    - Apply the rule to infer new facts.
    - Add the inferred facts to the working memory.

Example Rule: If $A \wedge B \to C$, and A, B are true, then infer C.

10.11 Backward Chaining

Backward chaining is a goal-driven approach that starts with a hypothesis and works backward to determine if the known facts support the hypothesis.

Algorithm:

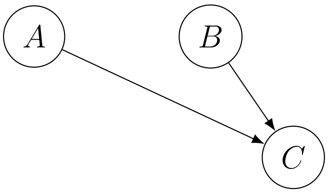
1. Start with the goal or query.
2. If the goal is a known fact, succeed.
3. If not, find rules that can infer the goal.
4. Recursively apply backward chaining on the premises of the rules.
5. If all premises are proven true, the goal is true.

10.12 Comparison

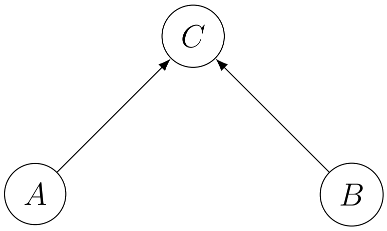
- **Forward Chaining:** Efficient for discovering all possible conclusions.
- **Backward Chaining:** Efficient for answering specific queries.

10.13 Graphical Representation

Forward Chaining Graph



Backward Chaining Graph



11 Planning under Uncertainty

> *”In which we see how an agent can take advantage of the structure of a problem to efficiently construct complex plans of action.”*
>
> — Stuart Russell, Peter Norvig

11.1 Markov Decision Process (MDP)

A **Markov Decision Process (MDP)** provides a mathematical framework for modeling decision-making where outcomes are partly random and partly under the control of a decision-maker. It is widely used in fields such as reinforcement learning, operations research, and robotics.

Definition

An MDP is defined as a tuple (S, A, P, R, γ), where:

- S: A finite set of states, representing the environment's possible configurations.
- A: A finite set of actions available to the agent.
- P: A state transition probability function $P(s'|s,a)$, specifying the probability of transitioning to state s' from state s when action a is taken.
- R: A reward function $R(s,a,s')$, which specifies the immediate reward received after transitioning from state s to state s' by taking action a.
- γ: A discount factor, $0 \leq \gamma \leq 1$, which determines the importance of future rewards.

11.2 The Bellman Equation (Value Iteration Algorithm)

The Bellman equation is a key concept in MDPs, used to compute the optimal policy and value functions. The value function $V(s)$ represents the expected cumulative reward starting in state s and following an optimal policy π. It is defined as:

$$V(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V(s') \right]$$

The optimal policy $\pi^*(s)$ can be derived by choosing the action a that maximizes the expected cumulative reward:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V(s') \right]$$

Example: Solving for $A3$ and $B3$

We are solving for the values of the states $A3$ and $B3$ in the following stochastic 4x3 grid. The transitions are stochastic:

- The intended action succeeds with probability $p = 0.8$.

- Other transitions (e.g., unintended moves to adjacent cells) occur with probability $1 - p = 0.2$.
- The discount factor is set to $\gamma = -3$, meaning future rewards are penalized more heavily.

Grid Representation

$$\begin{bmatrix} 0 & 0 & A3 & +100 \\ 0 & \text{Blocked} & B3 & -100 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- $+100$ represents the goal state (high reward).
- -100 represents the penalty state (high negative reward).
- Blocked represents an obstacle that cannot be traversed.
- $A3$ and $B3$ are the states we want to solve for.

11.3 Bellman Equation

The value for a state $V(s)$ is computed using the Bellman equation:

$$V(s) = \max_{a \in A} \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V(s') \right]$$

where:

- $\gamma = -3$: Negative discount factor.
- $R(s,a,s')$: Reward for transitioning to s'.
- $P(s'|s,a)$: Transition probabilities.

Example Calculation for $A3$

Let us compute $V(A3)$ with $\gamma = -3$. The possible actions are *up*, *down*, *left*, and *right*. Using the Bellman equation:

$$V(A3) = \max \left[0.8 \cdot \Big(R + \gamma V(A4) \Big) + 0.1 \cdot \Big(R + \gamma V(A2) \Big) + 0.1 \cdot \Big(R + \gamma V(B3) \Big) \right]$$

Substitute $\gamma = -3$, $V(A4) = +100$, $V(B3) = -100$, and assume $V(A2) = 0$ for simplicity:

$$V(A3) = \max \left[0.8 \cdot (0 - 3 \cdot 100) + 0.1 \cdot (0 - 3 \cdot 0) + 0.1 \cdot (0 - 3 \cdot -100) \right]$$

$$V(A3) = \max \left[0.8 \cdot (-300) + 0 + (30) \right]$$

$$V(A3) = \max \Big[-240 + 30 \Big] = -210$$

Example Calculation for $B3$

From $B3$, the possible actions are similar. Using the Bellman equation:

$$V(B3) = \max\left[0.8 \cdot \Big(R + \gamma V(A3)\Big) + 0.1 \cdot \Big(R + \gamma V(B2)\Big) + 0.1 \cdot \Big(R + \gamma V(B4)\Big)\right]$$

Substitute $\gamma = -3$, $V(A3) = -210$, $V(B4) = -100$, and assume $V(B2) = 0$:

$$V(B3) = \max\left[0.8 \cdot (0 - 3 \cdot -210) + 0.1 \cdot (0 - 3 \cdot 0) + 0.1 \cdot (0 - 3 \cdot -100)\right]$$

$$V(B3) = \max\left[0.8 \cdot 630 + 0 + 30\right]$$

$$V(B3) = \max\Big[504 + 30\Big] = 534$$

Results

The computed values for the states are:

$$V(A3) = -210, \quad V(B3) = 534$$

These values indicate how future penalties/rewards are amplified or diminished with the negative discount factor.

Partially Observable Markov Decision Processes (POMDPs)

A **Partially Observable Markov Decision Process (POMDP)** extends the Markov Decision Process (MDP) framework to scenarios where the agent does not have full knowledge of the current state. Instead, the agent must rely on indirect observations to infer the state.

Definition

A POMDP is defined as a tuple:

$$(S, A, P, R, \Omega, O, \gamma)$$

where:

- S: A finite set of states.
- A: A finite set of actions.
- $P(s'|s, a)$: Transition probabilities, as in MDPs.
- $R(s, a)$: Reward function.
- Ω: A finite set of observations.
- $O(o|s', a)$: Observation probabilities, representing the likelihood of observing o after transitioning to state s' with action a.
- γ: Discount factor.

Key Features

Belief States:

1. Since the agent does not know the exact state, it maintains a *belief state*, which is a probability distribution over all possible states.

2. For example, if $S = \{s_1, s_2, s_3\}$, a belief state might be $b = [0.2, 0.5, 0.3]$, meaning there is a 20% chance the agent is in s_1, 50% in s_2, and 30% in s_3.

Optimal Policy:

1. The goal is to find an optimal policy that maps belief states to actions in order to maximize expected cumulative rewards.

2. Unlike MDPs, where policies depend on states, POMDP policies depend on belief states.

Applications

POMDPs are used in scenarios where full observability is impractical, such as:

- **Robotics:** Navigating a partially observable environment.
- **Healthcare:** Diagnosing a disease based on uncertain symptoms and test results.
- **Speech Recognition:** Mapping uncertain acoustic signals to linguistic meanings.

Challenges

POMDPs are computationally more complex than MDPs due to:

- The need to maintain and update belief states.
- The exponentially larger space of possible belief states compared to regular states.
- Solving POMDPs optimally is often intractable for large problems.

Simplified Solution Approaches

While solving a POMDP optimally is challenging, several approximate methods exist, such as:

- **Point-Based Value Iteration (PBVI):** An approximate algorithm for solving POMDPs by focusing on a finite set of belief points.
- **Monte Carlo Sampling:** Using random samples to approximate the optimal policy.