CAB320 - Artificial Intelligence Study Notes

Jaden Ussher

July 25, 2024

Contents

1 Week 2 - Discrete Math Tools

1.1 Key Discrete Math Concepts for AI

- Recurrence relations and recursive functions
- Graphs and trees
  - Data structure, abstraction, object-oriented programming
- Graph properties
  - Directedness, paths, connectivity, connected components, cycles, roots, sinks
- Dijkstra's algorithm
  - Computation of the shortest paths from a source to all other vertices

1.2 Graphs and Their Importance in AI

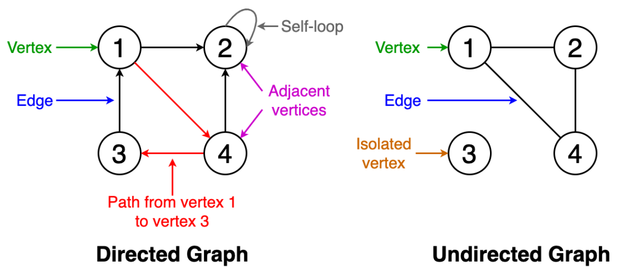


Figure 1: Graph Basics

- A graph G is a pair (V, E) where V is a set of vertices and E is a set of pairs of vertices called edges.
- Graphs do not need a visual representation to exist.
- Directed graphs (digraphs) have ordered edges, called arcs.
- Vertices are also known as nodes.
- In AI, graphs are used for:
  - Planning problems: Finding a "good" sequence of actions can be reduced to finding a "good" path in an associated state graph.
  - Knowledge representation: Social networks, scene representation, protein folding, etc.

Graph Representations

Adjacency Matrix

- For undirected graphs: A symmetric matrix where $A[i][j] = 1$ if there is an edge between vertices i and j, otherwise 0.
- For directed graphs: $A[i][j] = 1$ if there is an arc from i to j.

Vertex-Arc Incidence Matrix

- Represents the incidence of vertices and arcs.

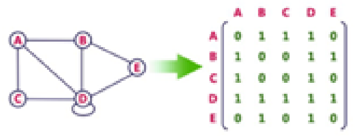


Figure 2: Undirected Graph Matrix

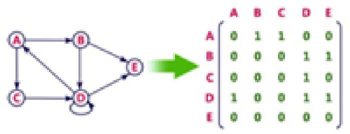


Figure 3: Directed Graph Matrix

Adjacency List

- Use dictionaries in Python where keys are vertices and values are lists of neighbors.
- Efficient for sparse graphs.

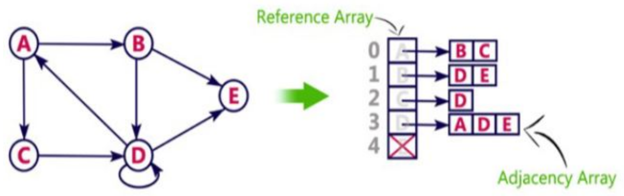


Figure 4: Adjacency List

Graph Properties

- Directedness: Graphs can be directed or undirected.
- Paths and connectivity:
  - A path is a sequence of vertices connected by edges.
  - Connectivity determines if there is a path between every pair of vertices.
  - A connected component is a maximal connected subgraph.
- Cycles: A path that starts and ends at the same vertex without repeating any edge or vertex.
- In-degree and Out-degree:
  - In-degree: Number of incoming arcs to a vertex.
  - Out-degree: Number of outgoing arcs from a vertex.

Special Graphs

Clique

- A clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent.
- Maximal clique: A clique that cannot be extended by including one more adjacent vertex.

Interval Graph

- Vertices represent intervals, and there is an edge between two vertices if their corresponding intervals intersect.
- Always contains a vertex whose neighbors form a clique.

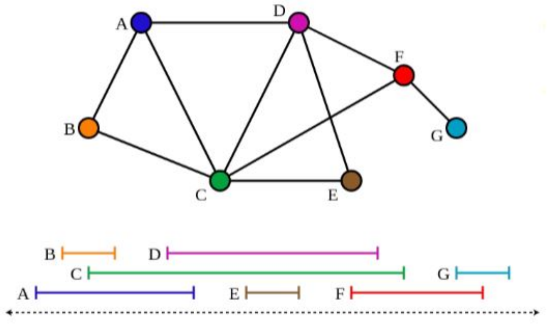


Figure 5: Interval Graph

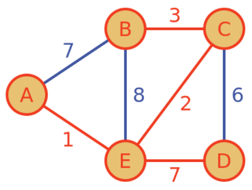
Trees

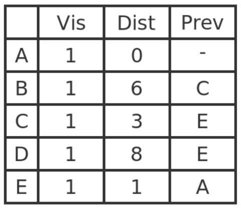
- A tree is a connected, acyclic undirected graph. Equivalent conditions for trees:
  - Connected and acyclic.
  - Adding any edge creates a cycle.
  - Removing any edge disconnects the graph.
  - Unique simple path between any two vertices.

1.3 Dijkstra's Algorithm

- Purpose: Find shortest paths from a source vertex to all other vertices in a weighted graph.
- Method:
  - Initialize distances from the source to all vertices as infinity except the source itself, which is zero.
  - Use a priority queue to repeatedly extract the vertex with the minimum distance.
  - Update the distances of the adjacent vertices.

Example 1





	Vis	Dist	Prev
A	1	0	-
B	1	6	C
C	1	3	E
D	1	8	E
E	1	1	A

Figure 6: Dijkstra's Algorithm Visual Example

Example 2

```
Graph: (V, E) with weights
    V = {A, B, C, D}
    E = {(A, B, 1), (A, C, 4), (B, C, 2), (B, D, 5), (C, D, 1)}
Steps:
1. Initialize distances: dist[A]=0, dist[B]=, dist[C]=, dist[D]=
2. Extract A: dist[A]=0
    Update dist[B]=1, dist[C]=4
3. Extract B: dist[B]=1
    Update dist[C]=3, dist[D]=6
4. Extract C: dist[C]=3
    Update dist[D]=4
5. Extract D: dist[D]=4
Result: Shortest paths from A
    A -> B -> C -> D with distances 0, 1, 3, 4 respectively.
```

Correctness of Dijkstra's Algorithm

- **Invariant Hypothesis**:
  - For each vertex v, dist$[v]$ is an upper bound of the cost of a cheapest path from the source to v.
  - If v has been removed from the priority queue, dist$[v]$ is the cost of the cheapest path.
- Proof by induction on the size of $V - L$ (where L is the set of unfinalized vertices).
- **Lemma**: If T is the tree of finalized vertices, any non-finalized vertex a adjacent to T has the cheapest path via its parent.
- **Contradiction**:
  - If there is a strictly cheaper path, it would imply incorrect extraction order, violating the algorithm's logic.

1.4 Week 2 - Prac Questions

Navigation System Exercise

Question: Most current navigation systems generate global, metric representation of the environment with either obstacle-based grid maps or feature-based metric maps. While suitable for small areas, global metric maps have inefficiencies of scale. Arguably, topological mapping is more efficient as it concisely represents a partial view of the world as a graph. In this Exercise, you will show that when the graph is without cycles, a map is superfluous for navigation purposes.

When following the instructions of a navigation GPS device, the driver of a car receives instructions of the form "at the next roundabout, take the third exit". This command format is well suited for logging the itinerary followed by a mobile agent in an environment that has a graphical topology like a road-network or the corridors of a building.

The figure below illustrates such an environment. The agent/robot traverses the graph following its edges and using its vertices as roundabouts. In order to indicate the direction the robot is facing on an edge, we specify the location of the robot by an arc. We adhere to the standard terminology of Graph Theory, and reserve the term arc for an edge that has been given an orientation.

The blue arrow on the left of arc ab represents the position of the robot going from Node a to Node b.

The navigational route instructions from the arc $a \to b$ as the starting position, to the arc labeled $h \to i$ as the destination, would sound as follows if told by a GPS device:

1. "drive to the next the intersection"
2. "turn first left"
3. "drive to the next the intersection"
4. "turn first left"
5. "drive to the next the intersection"
6. "turn second left"
7. "drive to the next the intersection"
8. "arrive at your destination"

A less verbose representation would code the route into the integer sequence $(1, 1, 2)$.

Observe that when arriving at a node with n edges, the commands c and c' have the same effect if and only if the difference $c - c'$ is a multiple of n.

Questions and Answers

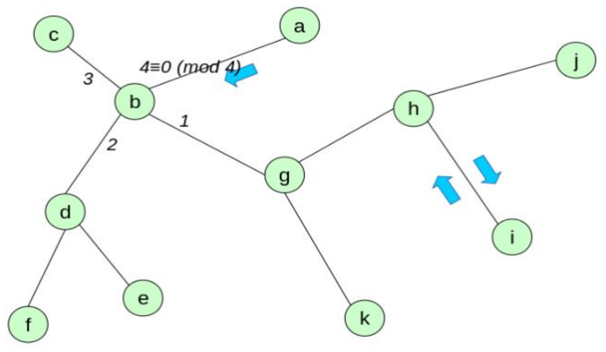


Figure 7: Interval Graph

What is the effect of the command $c = 0$?

Answer: The effect of the command $c = 0$ is to make a U-turn. This is because $c = 0$ means you don't turn at all, effectively continuing on the current path, which results in a U-turn back along the same edge you arrived from.

Compare the effect of the commands $-c$ and $n - c$.

Answer: The commands $-c$ and $n - c$ have the same effect. In a node with n edges:

- A command c means turning c edges clockwise.
- A command $-c$ means turning c edges counterclockwise.
- A command $n - c$ means turning c edges counterclockwise in a cyclic manner.

Thus, in a graph with n edges, $-c$ and $n - c$ are equivalent.

What are the possible commands for a U-turn?

Answer: The possible commands for a U-turn are multiples of n, where n is the degree of the node. This is because turning n edges clockwise or counterclockwise brings you back to the same edge, resulting in a U-turn.

What happens at a leaf node (node with exactly one incident edge)? Does the value of c matter at a leaf?

Answer: At a leaf node, the value of c is irrelevant as any value triggers a U-turn. This is because a leaf node has only one incident edge, so there is no choice but to return along the same edge.

Given a forward route $(c_1, c_2, \ldots, c_k)$ from arc α to arc β, provide a formula for a return route from β to α.

Answer: The formula for a return route is $(-c_k, \ldots, -c_2, -c_1)$, which can also be written as $(n_k - c_k, \ldots, n_2 - c_2, n_1 - c_1)$ considering the cyclic nature of the graph. This is because to return to the starting point, you need to undo each command in reverse order.

2 Week 3 - Intelligence Agents & Uniformed Search

2.1 Intelligence Agents

2.2 Key Concepts

Agents and Environments

- Agents include humans, robots, softbots, thermostats, etc.
- The agent function maps from percept histories to actions.
- Agents interact with environments through sensors and actuators.

Vacuum-Cleaner World

- Percepts: Location and contents, e.g., [A, Dirty].
- Actions: Left, Right, Suck, NoOp.
- Example: A vacuum-cleaner agent must decide its actions based on its current percept.

Rationality

- A rational agent chooses any action that maximizes the expected value of the performance measure given the percept sequence to date.
- Rationality maximizes expected outcome while perfection maximizes actual outcome.
- Rational does not mean omniscient or perfect.

P.E.A.S. Framework

- To design a rational agent, we must specify the task environment, which consists of:
  - Performance measure
  - Environment
  - Actuators
  - Sensors
- Examples:
  - Automated taxi: Safety, streets, steering, video.
  - Internet shopping agent: Price, WWW sites, display, HTML pages.
  - Part-picking robot: Percentage of parts in correct bins, conveyor belt, jointed arm, camera.

Environment Types

- Deterministic vs. stochastic: Is the next state perfectly predictable?
- Static vs. dynamic: Does the environment change while the agent deliberates?
- Fully vs. partially observable: Does the agent have access to all relevant information?
- Single-agent vs. multi-agent: Are there other agents in the environment?
- Known vs. unknown: Does the agent understand the laws governing the environment?
- Episodic vs. sequential: Are past actions relevant to the current decision?
- Discrete vs. continuous: Are states and time intervals fixed or continuous?

Agent Types

- Simple reflex agents: Choose actions based on current percept.
- Reflex agents with state: Have memory or a model of the world's current state.
- Goal-based agents: Act to achieve specific goals.
- Utility-based agents: Maximize a utility function to measure performance.
- Learning agents: Improve their performance over time.

Examples of Agent Types

- Reflex Agents: Edge following robot.
- Model-Based Reflex Agent: Vacuum robot that counts the number of cells cleaned.
- Model-Based Goal-Based Agent: Sokoban agent.
- Model-Based Utility-Based Agent: Pacman player.
- General Learning Agent: Self-balancing robot.

2.3 Uniformed Search

Planning Agents

- Planning agents ask "what if" and make decisions based on hypothesized consequences of actions.
- They must have a model of how the world evolves in response to actions.
- Formulation of a goal and consideration of how the world would be is essential.
- Distinguish between optimal vs. complete planning and planning vs. re-planning.

State Types

Different types of states in planning and search problems are considered.

Search Problems

A search problem consists of:

- A state space.
- A successor function (with actions and costs).
- A start state and a goal test.

A solution is a sequence of actions (a plan) that transforms the start state to a goal state.

Example Problems

- Romania: Finding paths between cities.
- Vacuum world: A simple robotic agent cleaning a room.
- The 8-puzzle: Sliding puzzle game.
- The 8-queens problem: Placing 8 queens on a chessboard without threatening each other.
- Robotic assembly: Planning the sequence of actions for assembling parts.

2.4 Tree Search Algorithms

Tree search algorithms explore possible sequences of actions to find a solution.

2.5 Uninformed Search Strategies

Uninformed search strategies are methods that search the state space without any domain-specific knowledge. These include:

- Breadth-First Search (BFS)
- Uniform-Cost Search
- Depth-First Search (DFS)
- Depth-Limited Search
- Iterative Deepening Search

2.5.1 *Breadth-First Search (BFS)*

- Explores all nodes at the present depth level before moving on to nodes at the next depth level.
- Initial frontier: {A}
- Frontier after expanding A: {B, C}
- Frontier after expanding B: {C, D, E}
- Frontier after expanding C: {D, E, F, G}

Properties: Complete and optimal for unweighted graphs, but has high memory requirements.

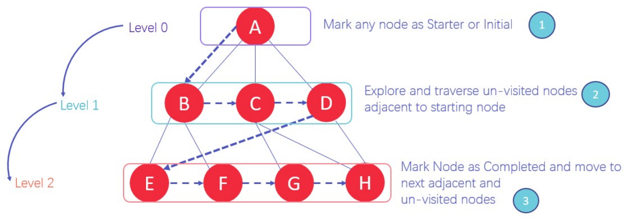

Level 0
Level 1
Level 2
A
B
C
D
E
F
G
H
Mark any node as Starter or Initial
1
Explore and traverse un-visited nodes adjacent to starting node
2
Mark Node as Completed and move to next adjacent and un-visited nodes
3

Figure 8: BFS Algorithm

2.5.2 *Uniform-Cost Search*

- Expands the least-cost unexpanded node.
- Similar to Dijkstra's algorithm for shortest paths.

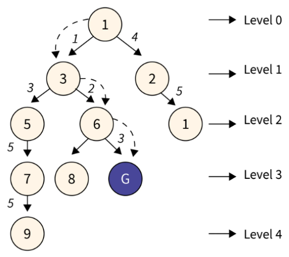


Figure 9: Uniform Cost Search Algorithm

2.5.3 *Depth-First Search (DFS)*

- Explores as far as possible along each branch before backtracking.
- Initial frontier: {A}
- Frontier after expanding A: {B, C}
- Frontier after expanding C: {B, F, G}
- Frontier after expanding G: {B, F, N, O}

Properties: May get stuck in infinite loops, not guaranteed to find the shortest path.

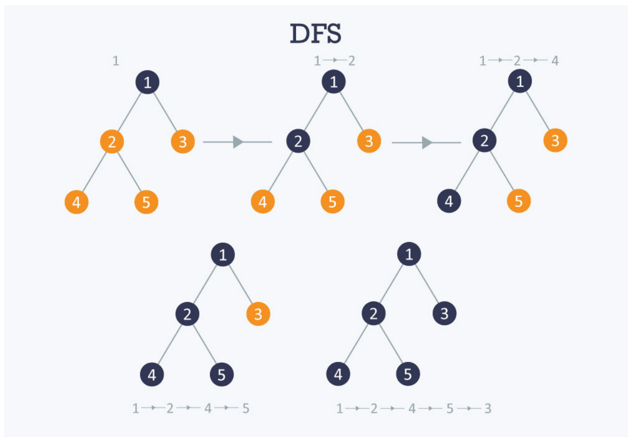


Figure 10: Depth First Search Algorithm

2.5.4 Depth-Limited Search

- Depth-first search with a limit on the depth.
- Useful for infinite depth or large search spaces.

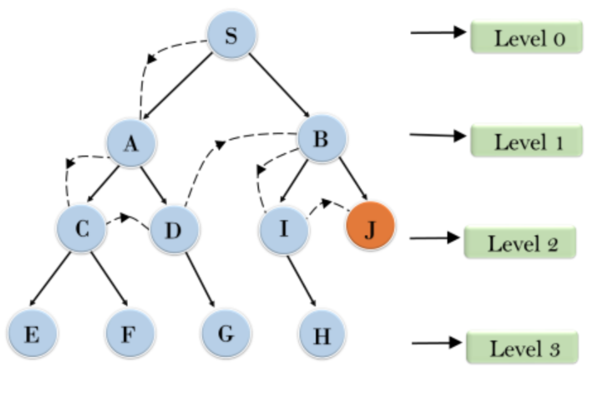


Figure 11: Depth Limited Search

2.5.5 Iterative Deepening Search

- Combines the benefits of BFS and DFS.
- Repeatedly applies DFS with increasing depth limits.
- Level 0: {A}
- Level 1: {B, C}
- Level 2: {D, E, F, G}

Properties: Complete and optimal like BFS, but with lower memory requirements.

Implementation Considerations

States vs. Nodes

- States represent physical configurations.
- Nodes represent states with additional information, such as parent nodes, actions, and path costs.

General Tree Search Algorithm

The general tree search algorithm explores nodes in a tree structure, using different strategies to manage the frontier.

Search Strategies

Different strategies impact the efficiency and effectiveness of finding a solution.

Graph Search

- Avoids expanding the same state multiple times.
- Maintains a frontier to explore and a set of explored nodes.

Example: Uniform-cost search on a subgraph of the Romania map.

Bidirectional Search

- Searches from both the start and goal states simultaneously.
- Can significantly reduce search time.

2.6 Week 3 - Mobile Robot Path Planning Problem

Problem: Two mobile robots $R1$ and $R2$ located in an $N \times N$ grid maze must meet. It does not matter in which cell of the maze they meet. At each time step, $R1$ and $R2$ simultaneously move in one of the following directions: {NORTH, SOUTH, EAST, WEST, STOP}. You must devise a plan which positions them together, somewhere in the grid, in as few time steps as possible. Passing each other does not count as meeting; they must occupy the same cell at the same time.

(a) Formally state this problem as a single-agent state-space search problem.

States: The state space can be expressed in set comprehension form as:

$$\{(x_1, y_1), (x_2, y_2) \mid x_1, x_2, y_1, y_2 \in \text{range}(N)\}$$

where (x_1, y_1) represents the coordinates of robot $R1$ and (x_2, y_2) represents the coordinates of robot $R2$.

Size of the state space:

$$N^4$$

This is because each robot can be in any of the $N \times N$ positions independently.

Branching factor of the search tree: Each robot has 5 possible actions: {NORTH, SOUTH, EAST, WEST, STOP}. Therefore, for two robots moving simultaneously, the branching factor is:

$$5 \times 5 = 25$$

This accounts for all possible combinations of movements for the two robots in a single time step.

Goal test function: The goal test function checks if both robots are in the same cell, i.e.,

$$\text{Goal}(x_1, y_1, x_2, y_2) \implies (x_1 = x_2) \text{ and } (y_1 = y_2)$$

This predicate returns true when the coordinates of $R1$ and $R2$ are identical, indicating that they have met.

(b) Give a non-trivial (different from zero) admissible heuristic for this problem. Assume that the cost of moving a single robot between two adjacent cells is 1.

An admissible heuristic for this problem needs to estimate the minimum cost to reach the goal from any given state without overestimating. A simple and admissible heuristic is:

$$h((x_1, y_1), (x_2, y_2)) = \frac{1}{2} \times (\text{Manhattan Distance}) = 0.5 \times (|x_1 - x_2| + |y_1 - y_2|)$$

Logic Behind the Heuristic:

- **Manhattan Distance:** The Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$. This distance represents the minimum number of moves one robot would have to make to reach the position of the other robot if the other robot remains stationary.
- **Dividing by 2:** Since both robots are moving simultaneously towards each other, dividing the Manhattan distance by 2 provides a lower bound on the number of time steps required for them to meet. This assumes optimal movement where both robots contribute equally to closing the distance between them.

Admissibility: The heuristic h is admissible because:

- It never overestimates the true cost to reach the goal. The actual cost will be at least half the Manhattan distance, as both robots can move towards each other simultaneously.
- It provides a non-zero estimate for any pair of distinct positions, ensuring that the search algorithm is guided efficiently towards the goal.

3 Week 4 - A* Tree Search and Search Heuristics

3.1 A* Search Algorithm

- Combines the strengths of Uniform Cost Search (UCS) and Greedy Search.
- Uses both path cost $g(n)$ and heuristic cost $h(n)$.
- Expands nodes based on the evaluation function $f(n) = g(n) + h(n)$.

*Key Properties of A**

- **Completeness:** A* is complete if the branching factor is finite and the cost of every step is greater than a positive constant.
- **Optimality:** A* is optimal if the heuristic used is admissible (i.e., it never overestimates the true cost).
- **Admissible Heuristics:** Heuristic $h(n)$ is admissible if $0 \leq h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal from node n.
- **Efficiency:** The efficiency of A* depends on the quality of the heuristic function. Better heuristics result in fewer nodes being expanded.

Example: A Search for Pathfinding*

- Consider finding a path in a grid from a start node S to a goal node G.
- **Path Cost $g(n)$:** The cost to reach node n from the start node S.
- **Heuristic Cost $h(n)$:** An estimate of the cost to reach the goal node G from node n. For instance, Manhattan distance or Euclidean distance.
- **Evaluation Function $f(n)$:** $f(n) = g(n) + h(n)$.
- A* search explores nodes in order of increasing $f(n)$ values.

Algorithm Steps

1. Initialize the open list with the start node S.
2. Initialize the closed list as empty.
3. Loop until the open list is empty or the goal node is found:
   (a) Remove the node n with the lowest $f(n)$ from the open list.
   (b) If n is the goal node G, reconstruct the path from S to G.
   (c) Generate successors of node n.
   (d) For each successor:
      - Calculate $g(\text{successor})$ and $f(\text{successor})$.
      - If the successor is not in the open list or has a lower f value, add it to the open list.
   (e) Add node n to the closed list.

3.2 Search Heuristics

- **Definition:** A heuristic is a function that estimates the cost to reach the goal from a given state.
- **Purpose:** Heuristics guide the search process, making it more efficient by prioritizing nodes that are more likely to lead to the goal.

Properties of Heuristics

- **Admissibility:** A heuristic is admissible if it never overestimates the cost to reach the goal.
- **Consistency (Monotonicity):** A heuristic is consistent if, for every node n and every successor n' of n, the estimated cost to reach the goal from n is no greater than the cost of getting to n' plus the estimated cost from n' to the goal.

Examples of Heuristics

- **Manhattan Distance:** Used for grid-based pathfinding where movement is restricted to horizontal and vertical steps.
$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$
- **Euclidean Distance:** Used for grid-based pathfinding where movement can be in any direction.
$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
- **Example in Pathfinding:**
  - In a grid, if the current node is at $(2, 3)$ and the goal node is at $(5, 7)$:
  - Manhattan distance heuristic: $h(n) = |2 - 5| + |3 - 7| = 3 + 4 = 7$.
  - Euclidean distance heuristic: $h(n) = \sqrt{(2 - 5)^2 + (3 - 7)^2} = \sqrt{9 + 16} = \sqrt{25} = 5$.

Heuristics for Specific Problems

- **Sliding Puzzle:** Number of tiles out of place, total Manhattan distance of tiles from their goal positions.
- **Traveling Salesperson Problem (TSP):** Minimum spanning tree (MST) heuristic, nearest neighbor heuristic.

4 Week 5 - A* Graph Search & Informed Search Analysis

4.1 A* Graph Search

- **Definition:** A* graph search extends the A* tree search to handle graphs where nodes can be revisited.
- **Key Features:**
  - Uses an evaluation function $f(n) = g(n) + h(n)$.
  - Maintains both open and closed lists to keep track of visited nodes and avoid re-expansion.

Algorithm Steps

1. Initialize the open list with the start node S.
2. Initialize the closed list as empty.
3. Loop until the open list is empty or the goal node is found:
   (a) Remove the node n with the lowest $f(n)$ from the open list.
   (b) If n is the goal node G, reconstruct the path from S to G.
   (c) Generate successors of node n.
   (d) For each successor:
      - Calculate $g(\text{successor})$ and $f(\text{successor})$.
      - If the successor is not in the open list or has a lower f value, add it to the open list.
      - If the successor is already in the closed list with a higher cost, update its cost and parent and move it back to the open list.
   (e) Add node n to the closed list.

Handling Revisited Nodes

- Nodes can be revisited if a lower-cost path to them is found.
- Ensures that the most cost-effective paths are considered, avoiding suboptimal solutions.

4.2 Optimality of A*

- **Definition:** A* is optimal if it always finds the least-cost path to the goal, provided the heuristic is admissible.
- **Admissible Heuristic:**
  - A heuristic $h(n)$ is admissible if $0 \leq h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal from node n.
- **Consistency (Monotonicity):**
  - A heuristic is consistent if for every node n and every successor n' of n, the estimated cost to reach the goal from n is no greater than the cost of getting to n' plus the estimated cost from n' to the goal.

Proof of Optimality

- **Admissibility Implies Optimality:**
  - A* expands nodes in order of increasing $f(n)$.
  - For any goal node n, $f(n) = g(n) + h(n)$.
  - If h is admissible, $f(n)$ is an underestimate of the true cost, ensuring that the first goal node expanded is the optimal one.
- **Consistency Implies Admissibility:**
  - If h is consistent, it is also admissible.
  - Ensures that the cost estimate $f(n)$ does not decrease along a path, which guarantees optimality.

Impact of Heuristic on Optimality

- **Admissible Heuristics:**
  - Heuristics that never overestimate the true cost.
  - Guarantee that A* finds an optimal solution.
- **Consistent Heuristics:**
  - Heuristics that satisfy $h(n) \leq c(n, n') + h(n')$ for every edge (n, n').
  - Ensure that the evaluation function $f(n)$ is non-decreasing, maintaining optimality.

Example: A* Graph Search with Heuristic

- Consider a graph with nodes A, B, C, and goal node G.
- **Path Costs:**
  - $g(A) = 0$, $g(B) = 1$, $g(C) = 2$, etc.
- **Heuristic Estimates:**
  - $h(A) = 3$, $h(B) = 2$, $h(C) = 1$.
- **Evaluation Function:**
  - $f(A) = g(A) + h(A) = 0 + 3 = 3$
  - $f(B) = g(B) + h(B) = 1 + 2 = 3$
  - $f(C) = g(C) + h(C) = 2 + 1 = 3$
- A* will expand nodes in order of $f(n)$ values, ensuring the optimal path is found if the heuristic is admissible.

4.3 A* Search with Modified Heuristic

- A curious student observed that multiplying a heuristic h by 100 significantly improved performance times.
- Questions arose regarding the behavior of A* when using a scaled heuristic.

Natural Questions

- If h is an admissible heuristic:
  - What happens if we use $5h$ instead of h?
  - What happens if we use $64h$ instead of h?
  - How does the returned solution compare with an optimal solution in the worst case?

Effect of Scaling the Heuristic

- Let $h' = 5h$, where h is admissible.
- Consider $n' = \arg\min\{g(x) + h'(x)\}$ and let n be a node on an optimal path to the goal.
- For x in the frontier:
  - $g(n') + h'(n') \leq g(n) + h'(n)$
  - $g(n) + h'(n) = g(n) + 5h(n)$
  - $g(n) + 5h(n) \leq 5g(n) + 5h(n)$
  - $5g(n) + 5h(n) \leq 5g(n) + 5h^*(n)$
  - Thus, $g(n') + h'(n') \leq 5 \times \text{optimal\_cost}$

Justification of Steps

- By definition of n' as the node minimizing $g + h'$.
- By definition of h' as $5h$.
- $1 \leq 5$ is trivially true.
- h is admissible, so $h(n) \leq h^*(n)$.
- n is on an optimal path.

Implications

- Using $g + 5h$ instead of $g + h$ means the returned path costs at most $5 \times$ optimal_cost.
- A* with $g + 5h$ behaves more like a greedy search:
  - May find a goal more quickly.
  - Loses the guarantee of optimality.

4.4 Week 5 - Problems

4.4.1 Admissible Heuristics

Assuming that h_1 and h_2 are admissible, which of the following expressions are also guaranteed to be admissible?

1. $h_1 + h_2$ **No, consider $h_1 = h_2 = h^*$, where h^* is not smaller or equal to h^*.**
2. $h_1 \cdot h_2$ **No, same counterexample as for (i).**
3. $\max(h_1, 0.3 \cdot h_2)$ **Yes, both expressions are smaller than h^*.**
4. $\min(h_1, 3 \cdot h_2)$ **Yes, same reason as for (iii).**
5. $0.94 \cdot h_1 + 0.08 \cdot h_2$ **No, same counterexample as for (i).**

5 Week 6 - Machine Learning

What is Machine Learning

- A computer (Agent) observes (or is given) some data.
- It builds a Model based on the data.
- Uses the Model as a hypothesis about the world and a piece of software that can solve problems.
- Given representative examples (the training data) of a phenomenon, can we predict/infer this for a new, previously unseen example?
  - If this example is in our training data, this is just data retrieval (lookup table).
  - E.g., Given pictures of a cat (and other things), can we determine if a new picture is a cat?

Types of Machine Learning

- Three main types of feedback accompany the input:
  - **Supervised Learning:** The agent observes input-output pairs and learns a function that maps from input to output.
    - E.g., Output = label in classification. Input may be an image containing an object, and output a class (e.g., table, house, cup, etc.).
  - **Unsupervised Learning:** The agent learns patterns in the input without any explicit feedback.
    - E.g., Most common type is clustering.
  - **Reinforcement Learning:** The Agent learns from a series of reinforcement: rewards and punishments.

Types of Learning Problems

Supervised Learning

- **Classification** (label prediction): The output is one of a finite set of values (e.g., true/false, sunny/cloudy).
- **Regression** (function approximation): The output is a number (or continuous variable) (e.g., tomorrow's temperature).

Unsupervised Learning

- Clustering.

Reinforcement Learning

- Policy Learning (learning from experience).

Machine Learning Process

- The **training** step is usually conducted offline (except for online learning), to build a model.
- The **validation** step (if needed) is also usually conducted offline to tune parameters or hyperparameters.
- The **testing** step is also usually conducted offline, to test the performance of a machine learning model on new (previously unseen) data.
- The **inference** or **prediction** (or estimation) is conducted 'online': this is when we actually use the machine learning algorithm to make an inference (or prediction or estimation) of the output we want to evaluate, given the input.

Data

- "A machine learning algorithm can only be as good as its data."
- Data sets split (partition):
  - **Training** set: to train candidate models.
  - **Validation** set (used when parameters or hyperparameters need tuning): to evaluate the candidate models and choose the best one.

    - Other names: development set or dev set.
  - **Test** set: To do a final unbiased evaluation of the selected model.
- Training, validation, and test sets should be exclusive (no overlap).
- Warning: in some communities/circles, the terms validation set and test set may be used the opposite way.

Holdout Validation

- Held-out data for validation and testing.

Cross-Fold Validation

- Used to evaluate the generalization of the ML method, and sensitivity to the training/validation/test split.
- Useful to select the 'best' ML method (e.g., classifier).
- Can be useful especially if limited labeled data available.

Deep Learning

- Uses multiple layers (deep) artificial neural networks.
- Can (often) do the feature extraction automatically.

Classical ML Examples

Regression

- Linear Regression.
- Multiple Regression.
- Logistic Regression.
- Advanced/Non-parametric Regression.

Classification

- Naive Bayes Classifier.
- K-Nearest Neighbours.
- Support Vector Machines (SVM).
- Random Forest.

Supervised Learning

- **Outcome Measurement** Y (also called dependent variable, response, target): What we are trying to predict/infer/estimate.
- Vector of p **predictor measurements** X (also called inputs, regressors, covariates, features, independent variables):
  - In the **regression** problem, Y is quantitative (e.g., price, blood pressure).
  - In the **classification** problem, Y takes values in a finite, unordered set (e.g., survived/died, digit 0-9, safe for landing).
- We have training data $(x_1, y_1), \ldots, (x_n, y_n)$. These are observations (or examples, instances) of these measurements.
- Let us call x the input and y the output (measurement).
- Given a training data set $(x_1, y_1), \ldots, (x_n, y_n)$ of N examples of input-output pairs where each pair was generated by an unknown function $y = f(x)$, estimate a function h that best approximates the true function.
- The function h will allow us to estimate the output y for any new input x.
- h is a "hypothesis about the world", or a model of the data.
- NB: $(y_1, \ldots, y_N)$ is part of the so-called "ground truth", assuming those example outputs are correct/accurate.
- h is said to generalize well if it allows us to accurately predict y for a new (previously unseen) test data set.

Learning Objectives

- Accurately predict the output (e.g., labels) for new test cases (i.e., new inputs).
- Understand which inputs affect the outcome and how.
- Assess the quality of prediction/inferences.

Bias and Variance

- **Bias**: Inability for a machine learning method to capture the true relationship between the input and the output (x and y).
- **Variance**: Difference in fit (error between estimates and reality) between the datasets. Variability of results for a new dataset.

Regression

Sum-of-Squares Error Function

- The sum of squares measures the deviation of data points away from the mean value.
- A higher sum of squares indicates higher variability while a lower result indicates low variability from the mean.
- To calculate the sum of squares, subtract the mean from the data points, square the differences, and add them together.
- $E(w) = \frac{1}{2}\sum_{n=1}^{N}[y(x_n, w) - t_n]^2$
- $E(w)$ measures the discrepancy between the model $y(x, w)$ and the dataset $\{(x_1, y_1), \ldots, (x_n, y_n)\}$.
- Objective: find the vector of weights w that minimize the error $E(w)$ i.e., minimize the **loss function**.

Loss Function

- $E(w)$ is often called the Loss function Loss (h_w) for the hypothesis h_w (candidate function to approximate the true function f).
- This particular loss function (squared-errors loss function) is named $L2$.
- We want to minimize Loss(h_w) i.e., find $w* = \arg\min_w(\text{Loss}(h_w))$.
- The minimum occurs when the partial derivatives of Loss are zero.
- For a linear regression problem (fit a line, i.e., $M = 1$) there is a direct, analytical solution to this optimization problem.
- For the general case, we use an optimization algorithm, e.g., gradient descent.

Over-Fitting

- In **machine learning**, **overfitting** occurs when an algorithm fits too closely or even exactly to its training data.
- As a result, the model becomes **overly specialized** and **unable to make accurate predictions or conclusions** when faced with data other than the training data.
- Essentially, it's like memorizing the training data without truly understanding the underlying patterns.

Regularisation

- Penalize large coefficient values by adding an extra term to the loss function.

6 Week 7 - Classification

Classification

Classification involves categorizing data into predefined classes based on input features. It is a fundamental task in machine learning and pattern recognition.

6.0.1 Naïve Bayes Classifier

- **Definition:** A probabilistic classifier based on Bayes' theorem with strong independence assumptions between features.
- **Bayes' Theorem:**
$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$
- **Types:**
  - **Multinomial Naïve Bayes:** Used for discrete data, such as text classification.
  - **Gaussian Naïve Bayes:** Assumes that the features follow a normal distribution.
- **Example:** Email spam classification where each word in the email is considered a feature.

6.0.2 K-Nearest Neighbors (KNN)

- **Definition:** A non-parametric, instance-based learning algorithm that classifies a sample based on the majority class among its k-nearest neighbors in the feature space.
- **Algorithm Steps:**
  1. Calculate the distance between the query sample and all training samples.
  2. Identify the k-nearest neighbors to the query sample.
  3. Assign the class label that is most frequent among the k-nearest neighbors.
- **Common Distance Metrics:**
  - Euclidean distance
  - Manhattan distance
- **Example:** Classifying handwritten digits based on the pixel values of images.

Support Vector Machines (SVM)

- **Definition:** A supervised learning model that finds the hyperplane which best separates the classes in the feature space.
- **Key Concepts:**
  - **Hyperplane:** The decision boundary separating different classes.
  - **Support Vectors:** Data points that are closest to the hyperplane and influence its position.
  - **Margin:** The distance between the hyperplane and the nearest support vectors; SVM aims to maximize this margin.
- **Kernel Trick:** Allows SVM to perform non-linear classification by mapping input features into higher-dimensional spaces.
  - **Common Kernels:**
    - Linear kernel
    - Polynomial kernel
    - Radial Basis Function (RBF) kernel
- **Example:** Classifying images of cats and dogs based on pixel intensities.

6.1 Week 7 - Problems

Problem: A binary classifier is trained to determine if an input image contains a cat (CAT) or does not contain a cat (NO-CAT). This classifier is given an image I and its output is that this image does not contain a CAT (NO-CAT). This classifier is not perfect and we have some statistics on its errors, based on a previous evaluation. We know that 7% of images available containing a CAT will be classified as NO-CAT by the classifier. We also know that if the image does not have a cat, the probability that the classifier returns that it contains a cat is 0.08. Also, we consider that in the general population of all images that may be considered, it is estimated that 0.2% do contain a cat.

Given the above information, compute the probability that the classifier made an error in deciding the image does not contain a cat (i.e. provided a "false negative").

Solution:

Let us pose:

- ImCAT = 1: The image contains a cat,
- ImCAT = 0: The image does NOT contain a cat,
- ClassCAT = 1: The classifier says there is a cat,
- ClassCAT = 0: The classifier says there is NO cat.

We are looking for $P(\text{ImCAT} = 1 \mid \text{ClassCAT} = 0)$, i.e., the probability that the image in fact contains a cat, knowing that the classifier said NO CAT.

From the text, we know that:

$$\begin{aligned} P(\text{ClassCAT} = 0 \mid \text{ImCAT} = 1) &= 0.07 \\ P(\text{ClassCAT} = 1 \mid \text{ImCAT} = 0) &= 0.08 \\ P(\text{ImCAT} = 1) &= 0.2\% = 0.002 \\ P(\text{ImCAT} = 0) &= 1 - P(\text{ImCAT} = 1) = 0.998 \end{aligned}$$

Using Bayes' Rule, we can write:

$$P(\text{ImCAT} = 1 \mid \text{ClassCAT} = 0) = \frac{P(\text{ClassCAT} = 0 \mid \text{ImCAT} = 1) \cdot P(\text{ImCAT} = 1)}{P(\text{ClassCAT} = 0)}$$

We know the elements of the numerator. For the denominator, we can write:

$$P(\text{ClassCAT} = 0) = P(\text{ClassCAT} = 0 \mid \text{ImCAT} = 0) \cdot P(\text{ImCAT} = 0) + P(\text{ClassCAT} = 0 \mid \text{ImCAT} = 1) \cdot P(\text{ImCAT} = 1)$$

We have:

$$\begin{aligned} P(\text{ClassCAT} = 0 \mid \text{ImCAT} = 0) &= 1 - P(\text{ClassCAT} = 1 \mid \text{ImCAT} = 0) = 1 - 0.08 = 0.92 \\ P(\text{ClassCAT} = 0 \mid \text{ImCAT} = 1) &= 0.07 \\ P(\text{ImCAT} = 0) &= 0.998 \\ P(\text{ImCAT} = 1) &= 0.002 \end{aligned}$$

Thus,

$$P(\text{ClassCAT} = 0) = 0.92 \cdot 0.998 + 0.07 \cdot 0.002 = 0.91816 + 0.00014 = 0.9183$$

Now, applying Bayes' theorem:

$$P(\text{ImCAT} = 1 \mid \text{ClassCAT} = 0) = \frac{P(\text{ClassCAT} = 0 \mid \text{ImCAT} = 1) \cdot P(\text{ImCAT} = 1)}{P(\text{ClassCAT} = 0)} = \frac{0.07 \cdot 0.002}{0.9183} = \frac{0.00014}{0.9183} \approx 0.000152$$

So, the probability that the image actually contains a cat given that the classifier predicted no cat (false negative) is approximately 0.000152.

7 Week 8 - Classification Evaluation Metrics

Confusion Matrix

- A confusion matrix is a table used to describe the performance of a classification model.
- It shows the counts of actual versus predicted classifications.
- The matrix has four key components:
  - **True Positives (TP):** Correctly predicted positive instances.
  - **True Negatives (TN):** Correctly predicted negative instances.
  - **False Positives (FP):** Incorrectly predicted positive instances.
  - **False Negatives (FN):** Incorrectly predicted negative instances.
- Example Confusion Matrix for a binary classification:

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

Accuracy

- Accuracy measures the proportion of correctly classified instances (both true positives and true negatives) out of the total instances.
- **Formula:**
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$
- Limitation: Not suitable when the dataset is imbalanced.

Precision (Positive Predictive Value)

- Precision measures the proportion of correctly predicted positive instances out of all instances predicted as positive.
- **Formula:**
$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall (Sensitivity or True Positive Rate)

- Recall measures the proportion of correctly predicted positive instances out of all actual positive instances.
- **Formula:**
$$\text{Recall} = \frac{TP}{TP + FN}$$

F1 Score

- The F1 score is the harmonic mean of precision and recall.
- It balances the two metrics, providing a single score that represents both.
- **Formula:**
$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$
- Useful when you need to balance precision and recall, especially for imbalanced datasets.

Specificity (True Negative Rate)

- Specificity measures the proportion of correctly predicted negative instances out of all actual negative instances.
- **Formula:**
$$\text{Specificity} = \frac{TN}{TN + FP}$$

ROC Curve and AUC

- A Receiver Operating Characteristic (ROC) curve plots the true positive rate (recall) against the false positive rate (1-specificity) at various threshold settings.
- The Area Under the ROC Curve (AUC) measures the entire two-dimensional area underneath the entire ROC curve.
- AUC provides an aggregate measure of performance across all possible classification thresholds.
- **Interpretation:**
  - AUC ranges from 0 to 1.
  - AUC = 0.5 suggests no discrimination (random guessing).
  - AUC = 1 indicates perfect classification.

8 Week 9 - Dynamic Programming

Dynamic Programming

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations.

Weighted Interval Scheduling

- **Problem:** Given a set of intervals each with a start time, end time, and weight, select a subset of non-overlapping intervals such that the total weight is maximized.
- **Brute Force:**
  - Enumerate all possible subsets of intervals.
  - Check each subset for non-overlapping intervals.
  - Compute the total weight for each valid subset.
  - **Running Time:** $O(2^n)$, where n is the number of intervals.
- **Dynamic Programming Approach:**
  - **Earliest Finish Time First:** Sort intervals by their finish times.
  - Define $p(j)$ as the largest index $i < j$ such that interval i is compatible with j.
  - Define $OPT(j)$ as the maximum weight of a subset of intervals from the first j intervals.
  - Recurrence relation: $OPT(j) = \max(w_j + OPT(p(j)), OPT(j-1))$
  - **Running Time:** $O(n \log n)$ (due to sorting) and $O(n)$ for computing the optimal weight.

Memoization

- **Definition:** An optimization technique where you store the results of expensive function calls and reuse them when the same inputs occur again.
- Often used in dynamic programming to store the results of subproblems.
- Reduces the time complexity from exponential to polynomial in many cases.

Binary Choice

- Many dynamic programming problems involve making a binary choice at each step.
- Example: In the weighted interval scheduling problem, the choice is whether to include a particular interval or not.

Least Squares / Segmented Least Squares

- **Problem:** Given a set of points, fit a set of line segments that minimize the sum of the squared errors.
- **Dynamic Programming Approach:**
  - Define $E(i, j)$ as the error of fitting a single line segment to points i through j.
  - Define $OPT(j)$ as the minimum cost of a segmentation of points 1 through j.
  - Recurrence relation: $OPT(j) = \min_{i<j}(OPT(i) + E(i, j) + C)$, where C is the cost of introducing a new segment.
  - **Running Time:** $O(n^2)$ where n is the number of points.

Multiway Choice

- Problems where you have more than two choices at each step.
- Example: Finding the optimal binary search tree where you choose the root node from multiple options.

False Start

- A common issue in dynamic programming where an initial approach needs to be revised because it doesn't lead to an optimal solution.
- Example: Trying to solve the knapsack problem without considering item weights correctly.

Knapsack Problem

- **Problem:** Given a set of items each with a weight and value, determine the maximum value that can be obtained by selecting items such that the total weight does not exceed a given limit.
- **Dynamic Programming Approach:**
  - Define $V(i, w)$ as the maximum value achievable with the first i items and a weight limit w.
  - Recurrence relation: $V(i, w) = \max(V(i-1, w), V(i-1, w-w_i) + v_i)$
  - **Running Time:** $O(nW)$, where n is the number of items and W is the weight limit.

Sequence Alignment

- **Problem:** Align two sequences to maximize their similarity, considering possible insertions, deletions, and substitutions.
- **Dynamic Programming Approach:**
  - Define $S(i, j)$ as the score of the best alignment of the first i characters of one sequence with the first j characters of the other sequence.
  - Recurrence relation:
$$S(i,j) = \max \begin{cases} S(i-1, j-1) + \text{match/mismatch} \\ S(i-1, j) + \text{gap penalty} \\ S(i, j-1) + \text{gap penalty} \end{cases}$$
  - **Running Time:** $O(mn)$, where m and n are the lengths of the two sequences.

8.1 Week 9 Problem - Edit Distance (Levenshtein Distance) Calculation

Problem: Compute the Edit Distance from the string "SWIMMING" to the string "SWINGS" with the following costs:

- Match: 0
- Mismatch: 7
- Insert: 2
- Delete: 3

Dynamic Programming Table:

		S	*W*	*I*	*N*	*G*	*S*
	0	2	4	6	8	10	12
S	3	0	2	4	6	8	10
W	6	3	0	2	4	6	8
I	9	6	3	0	2	4	6
M	12	9	6	3	5	7	9
M	15	12	9	6	8	10	12
I	18	15	12	9	11	13	15
N	21	18	15	12	9	11	13
G	24	21	18	15	12	9	11

Edit Operations:

1. Match S and S
2. Match W and W
3. Delete I
4. Delete M
5. Delete M
6. Match I and I
7. Match N and N
8. Match G and G
9. Insert S

9 Week 10 - Reinforcement Learning

Reinforcement Learning (RL)

Reinforcement Learning (RL) is concerned with goal-directed learning and decision-making. The RL framework allows you to build agents that learn by interacting with their environment.

Key Concepts in RL

- **Agent:** The learner or decision-maker that interacts with the environment.
- **Environment:** Everything the agent interacts with, encompassing states, actions, and rewards.
- **State:** A representation of the current situation of the agent within the environment.
- **Action:** Choices the agent can make that affect the environment.
- **Reward:** Feedback from the environment used to evaluate the last action.

Agent-Environment Interaction

- The agent perceives the current state of the environment.
- Based on the state, the agent takes an action.
- The environment responds to the action and provides a new state and a reward.
- This interaction is continuous, allowing the agent to learn and adapt over time.

Policy

- A **policy** is a stochastic rule for selecting actions given states.
- Denoted as $\pi(a|s)$, the probability of taking action a in state s.

Return

- The **return** is the function of future rewards the agent tries to maximize.
- Typically defined as the cumulative discounted reward: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$
- γ is the discount factor, $0 \leq \gamma \leq 1$.

Episodic and Continuing Tasks

- **Episodic Tasks:** Tasks that have a clear starting and ending point.
- **Continuing Tasks:** Tasks that go on indefinitely without a clear endpoint.

Markov Decision Process (MDP)

- MDPs are used to formalize RL problems.
- **Transition Probabilities:** $P(s'|s,a)$ - the probability of transitioning to state s' given state s and action a.
- **Expected Rewards:** $R(s,a)$ - the expected reward for taking action a in state s.

Value Functions

- **State-Value Function for a Policy:** $V^\pi(s) = \mathbb{E}^\pi[G_t|S_t = s]$
- **Action-Value Function for a Policy:** $Q^\pi(s,a) = \mathbb{E}^\pi[G_t|S_t = s, A_t = a]$
- **Optimal State-Value Function:** $V^*(s) = \max_\pi V^\pi(s)$
- **Optimal Action-Value Function:** $Q^*(s,a) = \max_\pi Q^\pi(s,a)$

Optimal Value Functions

- The value functions corresponding to the optimal policy π^*.
- **Optimal Policy:** The policy that maximizes the expected return from every state.

Bellman Equations

- **Bellman Equation for State-Value Function:**

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a) + \gamma V^\pi(s')\right]$$

- **Bellman Equation for Action-Value Function:**

$$Q^\pi(s,a) = \sum_{s'} P(s'|s,a) \left[R(s,a) + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s',a')\right]$$

The Need for Approximation

- In large state or action spaces, storing and updating value functions exactly is impractical.
- Approximation methods, such as function approximation with neural networks, are used to generalize from limited data.

Learning Algorithms

SARSA

- **State-Action-Reward-State-Action (SARSA):**
  - An on-policy learning algorithm that updates the action-value function based on the action taken by the current policy.
  - It stands for State-Action-Reward-State-Action, indicating that the update considers the action taken in the next state.
  - **Update Rule:**

    $$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

    where:
    * $Q(s_t,a_t)$ is the current estimate of the action-value function for state s_t and action a_t.
    * α is the learning rate.
    * r_{t+1} is the reward received after taking action a_t.
    * γ is the discount factor.
    * $Q(s_{t+1},a_{t+1})$ is the action-value function for the next state s_{t+1} and the next action a_{t+1}.
  - SARSA is called an on-policy method because it uses the current policy to determine the next action a_{t+1}.

Q-Learning

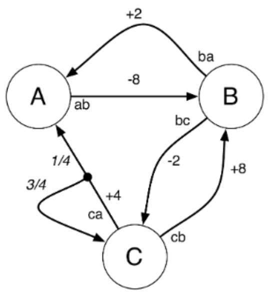
- **Q-Learning:**
  - An off-policy learning algorithm that updates the action-value function based on the maximum action-value for the next state, regardless of the policy's action.
  - It directly learns the optimal action-value function Q^*, independent of the policy being followed.
  - **Update Rule:**

    $$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right]$$

    where:
    * $Q(s_t,a_t)$ is the current estimate of the action-value function for state s_t and action a_t.
    * α is the learning rate.
    * r_{t+1} is the reward received after taking action a_t.
    * γ is the discount factor.
    * $\max_{a'} Q(s_{t+1},a')$ is the maximum action-value function for the next state s_{t+1} over all possible actions a'.
  - Q-Learning is called an off-policy method because it updates the action-value function using the best possible action (according to the current estimate) for the next state, not the action prescribed by the policy.

9.1 Week 10 Problems - Markov Decision Processes

The following digraph below shows a Markov Decision Process with a discount factor $\gamma = 0.5$. Upper case letters A, B, C represent states; arcs represent state transitions; lower case letters ab, ba, bc, ca, cb represent actions; signed integers represent rewards; and fractions represent transition probabilities.



Question 1

Define the state-value function $V^\pi(s)$ for a discounted MDP:
(a) State-Value Function $V(s)$:
- Definition: The expected return (sum of rewards) starting from state s and following policy π.

Question 2

Write down the bellman expectation equation for state-value functions:
(b) Bellman Expectation Equation for State-Value Functions:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^\pi(s')]$$

Question 3

Consider the uniform random policy $\pi''(s,a)$ that takes all actions from state s with equal probability. Starting with an initial value function of $V''(A) = V''(B) = V''(C) = 2$, apply one synchronous iteration of iterative policy evaluation (that is, one backup for each state) to compute a new value function $V^\#(s)$.

$V^\#(A)$	$V^\#(B)$	$V^\#(C)$
-7	1	7

$$\begin{aligned}
V^\#(A) &= -8 + 0.5 \cdot V^1(B) = -8 + 1 = -7 \\
V^\#(B) &= 0.5 \cdot (2 + 0.5 \cdot V^1(A)) + 0.5 \cdot (-2 + 0.5 \cdot V^1(C)) \\
&= 0.5 \cdot (2+1) + 0.5 \cdot (-2+1) = 1 \\
V^\#(C) &= 0.5 \cdot \left(4 + \frac{3}{4} \cdot 0.5 \cdot V^1(C) + \frac{1}{4} \cdot 0.5 \cdot V^1(A) + 0.5 \cdot (8 + 0.5 \cdot V^1(B))\right) \\
&= 0.5 \cdot \left(4 + \frac{3}{4} + \frac{1}{4}\right) + 0.5 \cdot (8+1) = 7
\end{aligned}$$

Question 4

Apply one iteration of greedy policy improvement to compute a new, deterministic policy $\pi_2(s)$

$$\pi(s) \leftarrow argmax_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$$

$$\pi(A) = ab \quad \text{as there is only one action available in } A$$

For $s = B,$ we have the returns:

$$ba : 2 + 0.5 \cdot V^2(A) = 2 - 3.5 = -1.5$$

$$bc : -2 + 0.5 \cdot V^2(C) = -2 + 3.5 = 1.5$$

Therefore $\pi(B) = bc$

For $s = C,$ we have the returns:

$$ca : 4 + \gamma \left(\frac{1}{4} \cdot V^2(A) + \frac{3}{4} \cdot V^2(C) \right) = 4 + 0.5 \cdot \left(-\frac{7}{4} + \frac{7 \cdot 3}{4} \right) = 5.75$$

$$cb : 8 + 0.5 \cdot V^2(B) = 8 + 0.5 = 8.5$$

Therefore, $\pi(C) = cb$

Question 5

*Define the optimal state-value function V *(s) for an MDP:*
Is the value of a state under an optimal policy. It is equal to the expected return for the best action from that state

Question 6

Write down te Bellman optimality Equation for state-value functions:

$$V^*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V^*(s')]$$

10 Week 11 - Artificial Neural Networks (ANNs)

Introduction to ANNs

- **Artificial Neural Networks (ANNs):** Computational models inspired by the human brain, designed to recognize patterns.
- Consist of layers of interconnected nodes (neurons).
- **Layers:**
  - **Input Layer:** Receives the input data.
  - **Hidden Layers:** Intermediate layers that transform the input into something the output layer can use.
  - **Output Layer:** Produces the final result.
- Each connection has an associated weight, and each neuron applies an activation function to its weighted sum of inputs.

Loss Functions

- **Loss Function:** A measure of how well the ANN's predictions match the actual data.
- Common loss functions:
  - **Mean Squared Error (MSE):**
    $$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$
    where y_i is the actual value and $\hat{y}_i$ is the predicted value.
  - **Cross-Entropy Loss:**
    $$\text{Cross-Entropy} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$
    Used for classification tasks.

Improving Loss Functions

- **Regularization:** Techniques to prevent overfitting by adding a penalty to the loss function.
  - **L1 Regularization:**
    $$L1 = \lambda \sum_j |w_j|$$
  - **L2 Regularization (Ridge):**
    $$L2 = \lambda \sum_j w_j^2$$
- **Dropout:** Randomly sets a fraction of neurons to zero during training to prevent over-reliance on certain paths.
- **Batch Normalization:** Normalizes the inputs of each layer to improve training speed and stability.

Activation Functions

- **Activation Function:** Determines the output of a neuron given an input or set of inputs.
- Common activation functions:
  - **Sigmoid:**
    $$\sigma(x) = \frac{1}{1+e^{-x}}$$
  - **Hyperbolic Tangent (Tanh):**
    $$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
  - **Rectified Linear Unit (ReLU):**
    $$\text{ReLU}(x) = \max(0, x)$$

- **Leaky ReLU:**

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

  where α is a small constant.

- **Softmax:** Used in the output layer for multi-class classification.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

11 Week 12 - CNNs and Unsupervised Learning

Activation Functions

Activation functions play a crucial role in neural networks by introducing non-linearity into the model. Non-linear activation functions are necessary because they allow neural networks to approximate complex functions, making them more powerful and capable of learning intricate patterns within data.

Sigmoid Function

The sigmoid function is one of the earliest activation functions used in neural networks. It's defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

The sigmoid function squashes the input values between 0 and 1, which makes it suitable for binary classification tasks where the output needs to be in the range [0, 1]. However, it suffers from the vanishing gradient problem, which can slow down or hinder the training process, especially in deep networks.

ReLU (Rectified Linear Unit)

ReLU is currently the most popular activation function used in deep neural networks. It's defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2}$$

ReLU is computationally efficient and helps alleviate the vanishing gradient problem, enabling faster training of deep networks. However, it can suffer from the problem of "dying" ReLU neurons, where neurons become inactive and stop learning.

Hyperbolic Tangent Function (Tanh)

The hyperbolic tangent function, tanh, is another widely used activation function. It's defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

Tanh squashes the input values between -1 and 1, which makes it suitable for classification tasks where the output needs to be in the range [-1, 1]. Like the sigmoid function, tanh also suffers from the vanishing gradient problem.

Back-propagation

Back-propagation is a foundational algorithm for training neural networks. It involves two main steps:

1. Forward Pass: In the forward pass, the input data is fed forward through the network. Each layer performs a linear transformation followed by an activation function to produce an output.
2. Backward Pass: In the backward pass, the errors between the predicted output and the actual output are calculated using a loss function. These errors are then propagated backward through the network using the chain rule of calculus to update the weights of the network, minimizing the loss function through gradient descent.

Back-propagation allows neural networks to learn from data by iteratively adjusting their parameters to minimize the difference between predicted and actual outputs.

11.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep neural networks, primarily used for analyzing visual imagery. They are inspired by the organization of the animal visual cortex and have proven to be highly effective in tasks such as image recognition, object detection, and image segmentation.

Key components of CNNs include:

- Convolutional Layers: Convolutional layers apply a set of filters to the input data, enabling the network to learn spatial hierarchies of features.
- Pooling Layers: Pooling layers downsample the feature maps generated by convolutional layers, reducing the spatial dimensions and the number of parameters in the network.
- Fully Connected Layers: Fully connected layers at the end of the network combine the features learned by convolutional layers and make predictions based on them.

CNNs have revolutionized the field of computer vision and are widely used in various applications, including autonomous driving, medical image analysis, and facial recognition.

Unsupervised Learning: Clustering Methods

Unsupervised learning is a type of machine learning where the model is trained on unlabeled data. Clustering is a common unsupervised learning technique used to group similar data points together.

K-means Clustering

K-means clustering is a centroid-based algorithm that aims to partition n observations into k clusters, where each observation belongs to the cluster with the nearest mean. The algorithm iteratively assigns data points to the nearest cluster centroid and updates the centroids based on the mean of the data points assigned to each cluster.

Hierarchical Clustering

Hierarchical clustering builds a tree of clusters, also known as a dendrogram. It does not require the number of clusters to be specified in advance and can be agglomerative (bottom-up) or divisive (top-down). Agglomerative hierarchical clustering starts with each data point as a separate cluster and merges the closest clusters iteratively, while divisive hierarchical clustering starts with all data points in one cluster and splits them recursively.

Hierarchical clustering is useful for exploratory data analysis and can reveal the hierarchical structure of the data.