Week 2

- **Uninformed Search - Introduction:** These strategies have no additional information about states beyond the problem definition. They can only generate successors and distinguish goal states. They are primarily differentiated by the order in which nodes are expanded.
- **Breadth-First Search (BFS):**
    - Expands the root node first, then all its successors, then their successors, and so on, working level by level.
    - Uses a FIFO (First-In, First-Out) queue for the frontier.
    - **Completeness:** Yes, if the shallowest goal is at a finite depth.
    - **Optimality:** Yes, if path cost is a non-decreasing function of depth (e.g., all actions have the same cost).
    - **Time Complexity:** O(b^d) (where 'b' is the branching factor and 'd' is the depth of the shallowest goal).
    - **Space Complexity:** O(b^d).
    - **Drawbacks:** High time and space complexity, with memory being a bigger problem than execution time for deep solutions.
- **Uniform-Cost Search (UCS):**
    - Expands the node with the lowest path cost (g(n)).
    - Uses a priority queue for the frontier, ordered by path cost.
    - The goal test is applied when a node is *selected* for expansion, not when *generated*, because a newly generated goal might be on a suboptimal path.
    - Handles varying step costs.
    - **Completeness:** Guaranteed if the cost of every step exceeds some small positive constant. It can get stuck in infinite loops with zero-cost actions.
    - **Optimality:** Yes, it expands nodes in order of their optimal path cost.
    - **Complexity:** Not easily characterized by 'b' and 'd', but rather by C* (cost of optimal solution) and ε (minimum step cost): O(b^(C*/ε)).
    - When all step costs are equal, it's similar to BFS but might do more work.
- **Depth-First Search (DFS):**
    - Always expands the deepest node in the current frontier.
    - Uses a LIFO (Last-In, First-Out) queue for the frontier.
    - **Completeness:**
        - Graph-search version: Yes, in finite state spaces.
        - Tree-search version: Not complete in infinite state spaces or with loops.
    - **Optimality:** No (it may find a deeper, suboptimal solution first).
    - **Time Complexity:**
        - Graph-search: Bounded by the size of the state space.
        - Tree-search: O(b^m) (where 'm' is the maximum depth of any node).
    - **Space Complexity:**
        - Graph-search: Not explicitly stated, but implies storing expanded nodes.
        - Tree-search: O(bm) (or O(m) with backtracking search and state modification).
    - **Advantages:** Significantly less memory required compared to BFS, making it practical for many AI problems.
- **Depth-Limited Search (DLS):**
    - A variation of DFS that introduces a predetermined depth limit 'l'. Nodes at depth 'l' are treated as if they have no successors.

    - **Completeness:** No, if 'l' < 'd' (the shallowest goal is beyond the limit).
    - **Optimality:** No, if 'l' > 'd'.
    - **Time Complexity:** O(b^l).
    - **Space Complexity:** O(bl).
    - DFS can be seen as DLS with an infinite limit.
- **Iterative Deepening Depth-First Search (IDS/IDDFS):**
    - Combines the benefits of DFS and BFS.
    - Gradually increases the depth limit (0, 1, 2, ... until a goal is found).
    - Repeatedly applies Depth-Limited Search with increasing limits.
    - **Completeness:** Yes, when the branching factor is finite.
    - **Optimality:** Yes, when path cost is a non-decreasing function of depth.
    - **Time Complexity:** O(b^d) (asymptotically the same as BFS, despite repeated work, because most nodes are at the deepest level).
    - **Space Complexity:** O(bd) (modest, like DFS).
    - **Advantages:** Often the preferred uninformed search method when the search space is large and the depth of the solution is unknown, due to its good balance of time and space efficiency.

Lecture 7

- **Bidirectional Search:** This technique runs two simultaneous searches—one forward from the initial state and one backward from the goal—to meet in the middle. It offers significant time and space complexity advantages (O(b^(d/2))) compared to standard BFS (O(b^d)), but its main weakness is the space requirement and the difficulty of searching backward when actions are not easily reversible or the goal state is complex (e.g., n-queens problem).
- **Comparing Uninformed Search Strategies:** A table (Figure 3.21) provides a comparison of various strategies based on criteria like completeness, time complexity, space complexity, and optimality. Strategies include Breadth-First, Uniform-Cost, Depth-Limited, Iterative Deepening, and Bidirectional search.
- **AND/OR Graphs:** These are hypergraphs used for problem decomposition. OR nodes represent choices where any successful path solves the problem, while AND nodes require all subproblems (descendants) to be solved for the parent node to be solved. The document outlines a procedure for traversing and updating estimates in AND/OR graphs to find the current best path.
- **Constraint Satisfaction Problem (CSP):** A CSP is defined by a set of variables (X), domains (D) for each variable, and constraints (C) that specify allowable combinations of values. A solution is a consistent, complete assignment of values to all variables that satisfies all constraints. The document explains how states are defined, the concept of consistent and complete assignments, and how DFS with forward checking can be used to solve CSPs.
- **Examples of CSPs:** Examples include the Map Coloring Problem (where adjacent regions cannot have the same color), Job Shop Scheduling, and the Cryptarithmetic Problem (e.g., SEND + MORE = MONEY, where letters are assigned unique digits 0-9). The document provides a detailed analysis of the Cryptarithmetic problem, demonstrating how constraints are used to deduce variable values. Other similar problems mentioned are Sudoku and the N-queens problem.

Lecture 8:
This document, "Lecture 8: Informed search strategies - 1," discusses informed search strategies in artificial intelligence, focusing on best-first search, greedy best-first search, and the A* algorithm, including memory-bounded heuristic search variations.

Here's a breakdown:

- **Informed Search Introduction:** Unlike uninformed strategies, informed search uses problem-specific knowledge to find solutions more efficiently. Best-first search is a general approach that uses an evaluation function `f(n)` to decide which node to expand, prioritizing the lowest cost estimate. Heuristic functions `h(n)` estimate the cost from a node to the goal.
- **Greedy Best-First Search:** This strategy expands the node closest to the goal by using `f(n) = h(n)`. While it can quickly find a solution, it is not optimal and can be incomplete, potentially leading to infinite loops in certain scenarios. An example using straight-line distance (h_SLD) to Bucharest in Romania is provided.
- **A* Algorithm:** This is a widely known best-first search algorithm that combines the cost to reach a node `g(n)` and the estimated cost to the goal `h(n)`, so `f(n) = g(n) + h(n)`. A* is both complete and optimal if the heuristic function `h(n)` is **admissible** (never overestimates the cost to the goal) and **consistent** (satisfies a form of the triangle inequality). Straight-line distance is an admissible and consistent heuristic. A* is also optimally efficient, meaning no other optimal algorithm is guaranteed to expand fewer nodes.
- **Memory-Bounded Heuristic Search:** Due to the high space complexity of A*, variants like Iterative-Deepening A* (IDA*) and Recursive Best-First Search (RBFS) are introduced to reduce memory requirements. IDA* uses f-cost as a cutoff, similar to iterative deepening. RBFS is a recursive algorithm that mimics best-first search using linear space by tracking the f-value of the best alternative path and backing up f-values.

Lecture 9:
This file, "lecture9.pdf," discusses informed search strategies, particularly focusing on memory-bounded heuristic search and heuristic functions.

Here's a summary of the key topics:

- **Memory-Bounded Heuristic Search:**
  - **RBFS (Recursive Best-First Search):** More efficient than IDA* but still suffers from node re-generation due to "mind changes" when the best path's f-value increases. It's optimal if the heuristic is admissible and has linear space complexity.
  - **IDA* and RBFS limitations:** Both suffer from using too little memory, leading to re-expanding states and potential exponential increases in complexity for redundant paths.
  - **MA* (Memory-bounded A*) and SMA* (Simplified MA*):** Algorithms that use all available memory. SMA* expands the best leaf until memory is full, then drops the worst leaf (highest f-value) and backs up its value to the parent. It's complete if a reachable solution exists (depth of goal < memory size) and optimal if an optimal solution is reachable. SMA* can become intractable for very hard

problems if it's forced to continually switch between many candidate paths.

- **Learning to Search Better:**
  - Introduces the concept of "metalevel state space" where states represent the computational state of a search program. Learning algorithms can learn from missteps in this metalevel space to avoid exploring unpromising subtrees and minimize total problem-solving cost.
- **Heuristic Functions:**
  - **8-Puzzle example:** Used to illustrate the nature of heuristics. It's a sliding tile puzzle with a branching factor of about 3 and 181,440 distinct reachable states for the 8-puzzle (10^13 for the 15-puzzle).
  - **Common heuristics for 8-puzzle:**
    - `h1`: Number of misplaced tiles (admissible).
    - `h2`: Sum of Manhattan distances of tiles from their goal positions (city block distance, also admissible). `h2` dominates `h1` and leads to greater efficiency.
  - **Effective Branching Factor (b*):** A way to characterize heuristic quality. A lower `b*` (closer to 1) indicates a more efficient heuristic. `h2` consistently shows a lower `b*` than `h1` and iterative deepening search.
  - **Domination:** If `h2(n) >= h1(n)` for all nodes, `h2` dominates `h1`, meaning A* with `h2` will never expand more nodes than with `h1`. Generally, heuristics with higher values are better, provided they are consistent and computationally inexpensive.
- **Generating Admissible Heuristics:**
  - **Relaxed Problem:** Heuristics can be derived from simplified versions of a problem with fewer restrictions on actions. The optimal solution cost for a relaxed problem is an admissible and consistent heuristic for the original problem. Examples for the 8-puzzle involve removing conditions on tile movement.
  - **Automatic Generation:** Programs like ABSOLVER can automatically generate heuristics using the relaxed problem method.
  - **Composite Heuristics:** Combining multiple admissible heuristics by taking the maximum value `h(n) = max{h1(n), ..., hm(n)}` results in an admissible and consistent heuristic that dominates its components.
  - **Subproblems (Pattern Databases):** Storing exact solution costs for subproblems (e.g., getting tiles 1, 2, 3, 4 into place in the 8-puzzle) can provide admissible heuristics. These are constructed by searching backward from the goal.
    - **Disjoint Pattern Databases:** If subproblems can be defined such that each move affects only one subproblem, their costs can be summed, leading to significant speedups (e.g., 10,000x for 15-puzzle, 1,000,000x for 24-puzzle compared to Manhattan distance).
- **Learning Admissible Heuristics from Experience:**
  - Solving many problem instances provides examples of states and their actual solution costs. A learning algorithm can use these to construct a function `h(n)` that predicts solution costs for new states.
  - **Feature-based learning:** Using relevant features (e.g., number of misplaced tiles, number of adjacent tiles out of place) in a linear combination like `h(n) = c1*x1(n) + c2*x2(n)`. The constants are adjusted to fit the data, but the

resulting heuristic is not necessarily admissible or consistent.

Lecture 10:

This document, "Lecture 10: Adversarial or game-playing search - 1," introduces adversarial search problems, also known as game-playing search, in competitive environments where two agents have conflicting goals.

Key topics covered include:

- **Two-player games:** Defined by an initial state, player turn, possible actions, transition model, terminal test, and a utility function (or payoff function). Chess is given as an example of a zero-sum game.
- **Game trees:** Represent the possible states and moves in a game. Tic-Tac-Toe is used as a simple example, illustrating how the utility values of terminal states are determined from the perspective of MAX (the player aiming for higher values).
- **Optimal decisions in games:** Discusses how to find the optimal strategy using the minimax value of each node in a game tree. The minimax value for MAX is the maximum of the minimum values of the successor states, assuming both players play optimally. The minimax algorithm recursively computes these values.
- **Minimax algorithm:** A depth-first exploration of the game tree. Its time complexity is O(b^m) and space complexity is O(bm) or O(m) depending on action generation. While impractical for large games, it forms the basis for other algorithms.
- **Optimal decisions in multiplayer games:** Extends the concept to more than two players, replacing single utility values with vectors of values. It also touches upon alliances and cooperation that can emerge from selfish behavior, especially in non-zero-sum games.
- **Alpha-beta pruning:** An optimization technique for minimax search that reduces the number of game states examined. It prunes branches of the game tree that cannot influence the final decision by maintaining `alpha` (the best choice found so far for MAX) and `beta` (the best choice found so far for MIN) values.

The document uses figures from "Artificial Intelligence: A Modern Approach" by Russel and Norvig to illustrate concepts like game trees and the alpha-beta pruning process.