Question 7: Consistency and Admissibility

Prove that consistency implies admissibility.

Definitions:

- A heuristic h(n) is **admissible** if for every node n, h(n) <= h*(n), where h*(n) is the true cost of the optimal path from n to a goal state. Also, h(goal) must be 0 for any goal state.
- A heuristic h(n) is **consistent** (or monotonic) if for every node n and every successor n' generated by an action a, the following condition holds: h(n) <= cost(n, a, n') + h(n'). This is a form of the triangle inequality.

Proof:
Let's consider an optimal path from node n to a goal state g. Let this path be n -> n_1 -> n_2 -> ... -> g.
By the definition of consistency, we can write a series of inequalities for the nodes along this path:

1. h(n) <= cost(n, a_1, n_1) + h(n_1)
2. h(n_1) <= cost(n_1, a_2, n_2) + h(n_2)
3. ...
4. h(n_k) <= cost(n_k, a_{k+1}, g) + h(g)

By definition, for a goal state g, h(g) = 0.
Now, we can substitute the inequality for h(n_1) from (2) into (1):
h(n) <= cost(n, a_1, n_1) + [cost(n_1, a_2, n_2) + h(n_2)]
h(n) <= cost(n, a_1, n_1) + cost(n_1, a_2, n_2) + h(n_2)

If we continue this substitution recursively along the entire path to the goal, we get:
h(n) <= cost(n, a_1, n_1) + cost(n_1, a_2, n_2) + ... + cost(n_k, a_{k+1}, g) + h(g)
h(n) <= sum(costs along the path from n to g)

Since this inequality holds for the optimal path from n to g, the sum of the costs is, by definition, h*(n).
Therefore, h(n) <= h*(n). This is the definition of admissibility.

Give an example of an admissible heuristic that is not consistent.

Consider the following state space:

- Nodes: Start (S), A, Goal (G)
- Edge Costs: cost(S, A) = 1, cost(A, G) = 4
- Optimal path from S to G is S -> A -> G with h*(S) = 5.
- Optimal path from A to G is A -> G with h*(A) = 4.

Let's define a heuristic h(n):

- h(S) = 3
- h(A) = 0
- h(G) = 0

1. Is the heuristic admissible?

- For S: h(S) = 3 <= h*(S) = 5. Yes.
- For A: h(A) = 0 <= h*(A) = 4. Yes.
- For G: h(G) = 0 <= h*(G) = 0. Yes.
  The heuristic is **admissible**.

2. Is the heuristic consistent?
We must check the consistency condition: h(n) <= cost(n, a, n') + h(n').

- Consider the transition from S to A:
  - h(S) = 3
  - cost(S, A) + h(A) = 1 + 0 = 1
  - The condition is 3 <= 1, which is **false**.

Because the consistency condition is violated for the transition from S to A, the heuristic is **not consistent**.

Question 9: Inadmissible Heuristic and Suboptimal Path

An inadmissible heuristic overestimates the true cost to the goal and can mislead A* into finding a suboptimal solution.

Example:
Consider the following graph:

- Start: S, Goal: G

- Path 1 (Optimal): S -> A -> G, with cost(S, A) = 2, cost(A, G) = 2. Total cost = 4.
- Path 2 (Suboptimal): S -> B -> G, with cost(S, B) = 3, cost(B, G) = 3. Total cost = 6.

Let's define an **inadmissible** heuristic h(n):

- h(A) = 5 (This is inadmissible because the true cost from A to G is 2, so 5 > 2).
- h(B) = 1 (This is admissible, 1 <= 3).
- h(S) = 0, h(G) = 0.

*A Search Walkthrough:**
A* expands nodes based on f(n) = g(n) + h(n), where g(n) is the cost from the start to n.

1. **Start at S.** Expand S. Successors are A and B.
    - Add A to fringe: f(A) = g(A) + h(A) = 2 + 5 = 7. Path: S->A.
    - Add B to fringe: f(B) = g(B) + h(B) = 3 + 1 = 4. Path: S->B.
    - Fringe: [(B, path S->B, f=4), (A, path S->A, f=7)]
2. **Expand B.** Pop B from the fringe (it has the lower f-value). The successor is G.
    - Add G to fringe: f(G) = g(G) + h(G) = (3 + 3) + 0 = 6. Path: S->B->G.
    - Fringe: [(G, path S->B->G, f=6), (A, path S->A, f=7)]
3. **Goal Found.** Pop G from the fringe. A* has found a path to the goal with cost 6. In many implementations of graph search, A* terminates here and returns the path S -> B -> G.

The algorithm returned a path with cost 6, while the optimal path S -> A -> G had a cost of 4. The inadmissible heuristic h(A) = 5 made the optimal path appear more expensive (f(A)=7) than the suboptimal path, causing the search to explore in the wrong direction and find the suboptimal goal first.

Question 10: Negative Edge Costs

Negative edge costs can lead to a paradoxical result where an optimal solution is infinitely long, preventing a search algorithm from terminating. This occurs when there is a **negative-cost cycle** in the state space.

Example:
Consider the following state space:

- Start: S, Goal: G

- Nodes A and B form a cycle.

Edge Costs:

- cost(S, A) = 3
- cost(A, B) = 2
- cost(B, A) = -4 (This creates the negative cycle)
- cost(B, G) = 5

Explanation of the Paradox:

1. **A simple path to the goal:** The path S -> A -> B -> G has a total cost of 3 + 2 + 5 = 10.
2. **Introducing the cycle:** An agent can instead travel from A to B and then back to A.
   - The cost of the cycle A -> B -> A is cost(A, B) + cost(B, A) = 2 + (-4) = -2.
3. **Infinite Path:** An agent can traverse this negative-cost cycle as many times as it wants to reduce its total path cost.
   - Path: S -> A -> (B -> A) -> B -> G. Cost = 3 + (-2) + 2 + 5 = 8.
   - Path: S -> A -> (B -> A) -> (B -> A) -> B -> G. Cost = 3 + 2*(-2) + 2 + 5 = 6.
   - Path with k loops: Cost = 10 + k*(-2).

As the number of loops (k) approaches infinity, the total path cost approaches negative infinity. There is no "lowest cost" path because for any path found, a longer path with a lower cost can be constructed by adding another loop through the negative cycle.

An algorithm searching for the optimal (lowest cost) solution would get trapped in this cycle, forever trying to find a better path. It would never reach the goal in finite time because the definition of an optimal solution is unattainable.

Question 11: Symbolic Search Problem

Problem Formulation:
You are asked to find a solution that is a sequence of at most five symbols from an alphabet of thousands.

- **States:** A state is a sequence of symbols. The start state is the empty sequence ().
- **Actions:** An action is to append a symbol from the alphabet to the current sequence.

- **Successor Function:** For a state s (a sequence), the successor function returns a set of new states {s + symbol_1, s + symbol_2, ...} for every symbol in the alphabet.
- **Goal Test:** A function is_solution(sequence) which returns True if the given sequence is a valid solution.
- **Path Cost:** The cost of a path is the length of the sequence, assuming each symbol has a cost of 1.

Algorithm Choice and Explanation:
The search tree for this problem is very wide (branching factor b is thousands) but very shallow (maximum depth d is 5).

- **Depth-First Search (DFS):** This would be a **poor choice**. With a branching factor in the thousands, DFS would pick one symbol and explore all sequences of length five starting with that symbol before ever trying a different first symbol. It would likely explore millions of irrelevant nodes down one deep branch before finding a solution.
- **Breadth-First Search (BFS):** This is a **good choice**. BFS explores layer by layer. It would check all sequences of length 1, then all of length 2, and so on. Since the solution is guaranteed to be at a shallow depth (<= 5), BFS is guaranteed to find the shortest possible solution sequence. The main drawback is memory, as the number of nodes at depth d is b^d. However, for a small d like 5, this may be manageable, and it is far superior to DFS's uninformed exploration.

How might you change the algorithm?
The best algorithm for this scenario would be **Iterative Deepening Search (IDS)**.

IDS combines the strengths of both DFS and BFS. It performs a series of depth-limited searches.

1. Run DFS with a depth limit of 1.
2. If no solution, run DFS with a depth limit of 2.
3. ...and so on, up to the maximum depth of 5.

Why IDS is better:

- **Finds Optimal Solution:** Like BFS, it finds the shortest solution because it explores depths incrementally.
- **Low Memory Usage:** Like DFS, its memory footprint at any time is only proportional to the current depth (O(bd)), not the total number of nodes at that depth (O(b^d)). This completely solves the memory issue of BFS with a large branching factor.

For a problem with a vast branching factor and a known shallow solution depth, IDS is the ideal algorithm.

Question 12: Iterative Deepening for Lowest Cost

Modification:
Standard Iterative Deepening Search (IDS) finds the shortest path by using a *depth* limit that increases with each iteration. To find the lowest *cost* path, we must instead use a *cost* limit. This algorithm is sometimes called *Iterative Deepening A (IDA*)**, though a heuristic is not strictly required.

The Algorithm (Iterative Deepening by Cost):

1. Initialize a cost_limit. A good starting value is the heuristic estimate of the start state's cost to the goal, h(start), or 0 if no heuristic is used.

2. Start an infinite loop:
   a. Set a variable min_next_cost to infinity. This will track the lowest cost of a path that was pruned.
   b. Perform a recursive, depth-first style search from the start state.
   c. During the search, for any node n with path cost g(n), if g(n) > cost_limit, prune this branch (do not explore its children). When pruning, update min_next_cost = min(min_next_cost, g(n)).
   d. If the search finds a goal state, return the path and its cost. This is the optimal solution.
   e. If the search completes without finding a goal, update
   the cost_limit to min_next_cost and start the loop again.

Information Needed:

1. The **cumulative cost g(n)** for the path to every node n.

2. A variable to store the **minimum cost of all pruned paths** in the current iteration, which will become the cost limit for the next iteration.

Proof of Optimality:
The algorithm is guaranteed to find the lowest-cost path.

1. **Ordered Search:** The algorithm explores the search space in iterations of non-decreasing cost. The cost_limit for each iteration is strictly greater than the cost_limit of the previous one (unless no paths exist).

2. **Completeness in Iteration:** Within a single iteration with a given cost_limit, the modified DFS is guaranteed to find any goal state reachable with a path cost less than or equal to that cost_limit.
3. **Finding the Optimal Path:** Let the optimal path have a cost C*.
    - Any iteration with cost_limit < C* will fail to find the goal and will set up a new, higher cost_limit for the next iteration.
    - Eventually, an iteration will run with a cost_limit >= C*. Let the first such iteration have a cost limit C_k (where C_k = C*).
    - In this iteration, the search is guaranteed to explore all paths up to cost C_k. Since an optimal path with cost C* exists, it will be found.
    - Because this is the *first* iteration where the cost limit is high enough to find *any* solution, the first solution found must be an optimal one. Any suboptimal solution would have a cost greater than C* and would only be found in a later iteration with a higher cost limit.