Assignment 1

AI (CSL7610) - Search Strategies & Constraint Satisfaction

Total Points: 70

Code Links: q1_search.py (Question 1) | q2_csp.py (Question 2)

Question 1: Search Strategies

Scenario: Lead AI Architect for a digital archive. A 3x3 shelf of rare manuscripts has been scrambled by a system glitch. Program a robotic sorter to restore order using various search strategies.

1.1 Problem Formulation (4 x 2.5 = 10 points)

a. State Representation (2.5 pts)

The state is represented as a **tuple of 9 elements** in row-major order, corresponding to the 3x3 grid. Each element is an integer (1-8) for a manuscript, or 'B' for the blank slot. Example: start state 1 2 3 / B 4 6 / 7 5 8 is stored as `(1, 2, 3, 'B', 4, 6, 7, 5, 8)`. Tuples are immutable and hashable, enabling O(1) membership tests in visited sets. Position is derived from index: row = index // 3, col = index % 3.

b. Actions (2.5 pts)

Four possible actions: **Up, Down, Left, Right** -- moving the blank in the corresponding direction (swapping it with the adjacent tile). An action is valid only if the resulting position is within bounds (0 <= row < 3, 0 <= col < 3). The `get_neighbors(state)` function generates all valid (action, new_state) pairs by attempting each move. The number of valid actions ranges from 2 (blank in corner) to 4 (blank in center).

c. Goal State (2.5 pts)

The goal state is `(1, 2, 3, 4, 5, 6, 7, 8, 'B')`, i.e., manuscripts in sequential order (1 2 3 / 4 5 6 / 7 8 B) with the blank in the bottom-right. The goal test is a simple tuple equality comparison: `state == GOAL_STATE`. This is O(1) amortized due to Python's hash caching for tuples.

d. Path Cost Function (2.5 pts)

Each move costs **1 unit of System Energy** (uniform cost). Thus g(n) = depth of node n = number of moves from start. The total path cost equals the solution length. For A*, f(n) = g(n) + h(n). Since costs are uniform, BFS guarantees finding the minimum-cost (optimal) solution.

1.2 Implementation of Search Algorithms

All algorithms below use the **exact same** state representation, action model, goal test, and path cost as defined in Section 1.1.

A. Uninformed Search (5 x 2 = 10 points)

a. Breadth-First Search (BFS)

BFS explores all states at depth d before depth d+1 using a FIFO queue. A visited set prevents revisiting any state (including root and parent). BFS guarantees finding the minimum number of moves since each move has uniform cost 1. Early goal testing checks neighbors before enqueuing.

```
def bfs(start, goal=GOAL_STATE):
    frontier = deque([start]); visited = {start}; came_from = {}
    while frontier:
        current = frontier.popleft()
        for action, neighbor in get_neighbors(current):
            if neighbor not in visited:
                came_from[neighbor] = (current, action)
                if is_goal(neighbor, goal):
                    return reconstruct_path(came_from, neighbor)  # SUCCESS
                visited.add(neighbor); frontier.append(neighbor)
    return None  # FAILURE
```

```
Algorithm: BFS | Result: SUCCESS | States Explored: 8
Path Length: 3 moves | Path: Right -&gt; Down -&gt; Right
Time: 0.000034 seconds
```

b. Depth-First Search (DFS)

DFS explores deep into the search tree using a LIFO stack. Two mechanisms prevent infinite paths: (1) a **depth limit of 50**, and (2) a **global visited set** that prevents revisiting root, parent, or any explored state. DFS is not optimal (found 11 moves vs. BFS's optimal 3) but uses less memory in typical cases.

```
def dfs(start, goal=GOAL_STATE, max_depth=50):
    stack = [(start, 0)]; visited = {start}; came_from = {}
    while stack:
        current, depth = stack.pop()
        if is_goal(current, goal): return reconstruct_path(came_from, current)
        if depth &gt;= max_depth: continue  # Depth limit prevents infinite paths
        for action, neighbor in get_neighbors(current):
            if neighbor not in visited:  # Prevents revisiting any state
                visited.add(neighbor); came_from[neighbor] = (current, action)
                stack.append((neighbor, depth + 1))
    return None
```

```
Algorithm: DFS | Result: SUCCESS | States Explored: 107,110
Path Length: 11 moves | Time: 0.193252 seconds
Path: Right-&gt;Right-&gt;Down-&gt;Left-&gt;Up-&gt;Right-&gt;Down-&gt;Left-&gt;Up-&gt;Right-&gt;Down
```

B. Informed Search (5 x 2 = 10 points)

Heuristics (used by Greedy, A*, and IDA*):

```
def h1_misplaced(state, goal):  # Count tiles not in goal position
    return sum(1 for i in range(9) if state[i] != 'B' and state[i] != goal[i])

def h2_manhattan(state, goal):  # Sum of Manhattan distances
    dist = 0
    for i in range(9):
        if state[i] != 'B':
            gi = goal.index(state[i])
            dist += abs(i//3 - gi//3) + abs(i%3 - gi%3)
    return dist
```

a. Greedy Best-First Search

Uses **only h(n)** to select nodes (ignores path cost g(n)). Expands the node appearing closest to the goal. Uses a min-heap priority queue keyed by h(n). Not guaranteed optimal, but often fast.

```
def greedy_best_first(start, goal, heuristic):
    frontier = []; heapq.heappush(frontier, (heuristic(start, goal), 0, start))
    visited = {start}; came_from = {}
    while frontier:
        _, _, current = heapq.heappop(frontier)
        if is_goal(current, goal): return reconstruct_path(came_from, current)
        for action, neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor); came_from[neighbor] = (current, action)
                heapq.heappush(frontier, (heuristic(neighbor, goal), cnt, neighbor))
```

```
Greedy (h1): SUCCESS | States: 4 | Path: 3 | Time: 0.000044s
Greedy (h2): SUCCESS | States: 4 | Path: 3 | Time: 0.000032s
```

b. A Search*

A* uses **f(n) = g(n) + h(n)**, combining path cost with heuristic. Optimal and complete with admissible heuristics. Expands the node with lowest f-value using a min-heap.

```
def a_star(start, goal, heuristic):
    frontier = []; g_score = {start: 0}
    heapq.heappush(frontier, (heuristic(start, goal), 0, start))
    visited = set(); came_from = {}
    while frontier:
        f, _, current = heapq.heappop(frontier)
        if current in visited: continue
        visited.add(current)
        if is_goal(current, goal): return reconstruct_path(came_from, current)
        for action, neighbor in get_neighbors(current):
            new_g = g_score[current] + 1  # Uniform cost = 1
            if neighbor not in visited and (neighbor not in g_score or new_g &lt; g_score[neighbor]):
                g_score[neighbor] = new_g; came_from[neighbor] = (current, action)
                heapq.heappush(frontier, (new_g + heuristic(neighbor, goal), cnt, neighbor))
```

```
A* (h1): SUCCESS | States: 4 | Path: 3 (optimal) | Time: 0.000023s
A* (h2): SUCCESS | States: 4 | Path: 3 (optimal) | Time: 0.000030s
```

C. Memory-Bounded & Local Search (5 x 2 = 10 points)

a. Iterative Deepening A (IDA*)*

IDA* combines A*'s optimality with DFS's memory efficiency. It performs iterative DFS with an f-cost threshold that increases each iteration to the minimum f-value exceeding the previous threshold. Space complexity is **O(d)** vs. A*'s O(b^d). Uses the same heuristics (h1, h2) and state representation as A*.

```
def ida_star(start, goal, heuristic):
    threshold = heuristic(start, goal); path_stack = [start]
    def search(g, threshold):
        node = path_stack[-1]; f = g + heuristic(node, goal)
        if f &gt; threshold: return f
        if is_goal(node, goal): return "FOUND"
        min_t = float('inf')
        for action, neighbor in get_neighbors(node):
            if neighbor not in set(path_stack):  # Cycle detection on path
                path_stack.append(neighbor)
                t = search(g + 1, threshold)
                if t == "FOUND": return "FOUND"
                min_t = min(min_t, t); path_stack.pop()
        return min_t
    while True:
        result = search(0, threshold)
        if result == "FOUND": return path_stack
        if result == float('inf'): return None
        threshold = result  # Increase threshold
```

```
IDA* (h1): SUCCESS | States: 8 | Path: 3 | Iterations: 1 | Time: 0.000046s
IDA* (h2): SUCCESS | States: 8 | Path: 3 | Iterations: 1 | Time: 0.000032s
```

b. Simulated Annealing

Local search that escapes local optima by occasionally accepting worse moves. **Energy:** Manhattan Distance (h2). **Cooling schedule:** T = T_{init} * cooling_rate^{iter}. **Acceptance:** Better moves always accepted; worse moves accepted with probability **P = $e^{-deltaE/T}$**. As T decreases, acceptance of worse moves decreases.

```
def simulated_annealing(start, goal, T_init=5000, cooling_rate=0.9995):
    current = start; T = T_init
    current_energy = h2_manhattan(current, goal)
    for i in range(max_iterations):
        if current_energy == 0: break  # Goal reached
        action, neighbor = random.choice(get_neighbors(current))
        delta_e = h2_manhattan(neighbor, goal) - current_energy
        if delta_e &lt;= 0 or random.random() &lt; math.exp(-delta_e / T):
            current = neighbor; current_energy += delta_e
        T *= cooling_rate  # Geometric cooling
```

```
SA (T0=5000, cool=0.9995, seed=42): SUCCESS | States: 17,397 | Time: 0.047s
Note: Stochastic -- some seeds may not converge (seed=123: FAILURE, energy=4)
```

D. Adversarial Search Extension (5 x 2 = 10 points)

a. Minimax Formulation

Game States: Same 9-element tuple. **MAX (Robotic Sorter):** Maximizes utility to reach goal. **MIN (System Glitch):** Minimizes utility to increase disorder. **Terminal:** Goal reached (utility=0) or depth limit. **Utility:** -Manhattan_Distance (0 at goal, more negative = more disorder).

```
def minimax(state, depth, is_max_player, goal):
    if is_goal(state, goal) or depth == 0:
        return -h2_manhattan(state, goal), None
    if is_max_player:
        best = float('-inf')
        for action, nbr in get_neighbors(state):
            val, _ = minimax(nbr, depth-1, False, goal)
            if val &gt; best: best = val; best_act = (action, nbr)
        return best, best_act
    else:  # MIN player
        best = float('inf')
        for action, nbr in get_neighbors(state):
            val, _ = minimax(nbr, depth-1, True, goal)
            if val &lt; best: best = val; best_act = (action, nbr)
        return best, best_act
```

```
Minimax (depth=6): Utility=-3 | Best: Right | Nodes: 766 | Time: 0.0012s
```

b. Alpha-Beta Pruning

Optimizes Minimax by maintaining bounds: **alpha** (MAX's guaranteed best) and **beta** (MIN's guaranteed best). When alpha >= beta, remaining children are pruned. Returns the same optimal result with fewer node evaluations.

```
def alpha_beta(state, depth, alpha, beta, is_max_player, goal):
    if is_goal(state, goal) or depth == 0:
        return -h2_manhattan(state, goal), None
    if is_max_player:
        best = float('-inf')
        for action, nbr in get_neighbors(state):
            val, _ = alpha_beta(nbr, depth-1, alpha, beta, False, goal)
            if val &gt; best: best = val; best_act = (action, nbr)
            alpha = max(alpha, best)
            if alpha &gt;= beta: break  # PRUNE
        return best, best_act
    else:
        best = float('inf')
        for action, nbr in get_neighbors(state):
            val, _ = alpha_beta(nbr, depth-1, alpha, beta, True, goal)
            if val &lt; best: best = val; best_act = (action, nbr)
            beta = min(beta, best)
            if alpha &gt;= beta: break  # PRUNE
        return best, best_act
```

```
Alpha-Beta (depth=6): Utility=-3 | Best: Right | Nodes: 546 | Pruned: 141
Time: 0.0009s | Node Reduction: 28.7% | Speedup: 1.33x
```

Efficiency Comparison:

Metric	Minimax	Alpha-Beta
Nodes Evaluated	766	546
Branches Pruned	0	141
Time (s)	0.0012	0.0009
Optimality	Yes	Yes (same result)
Node Reduction	-	28.7%

1.3 State Tracking & Performance

All algorithms maintain state tracking using the same tuple-based state structure to prevent revisiting states:

Visited Set (Explored/Closed): A Python set of state tuples storing all expanded states. Before adding a neighbor to the frontier, it is checked against this set. This enforces: (1) the root is never revisited (added to visited immediately), (2) the parent is never revisited (already in visited), and (3) no previously explored state is revisited. Tuples provide O(1) hash-based lookup.

Frontier (Not_Visited/Open): Contains generated but unexpanded states. Structure varies: deque (BFS), stack (DFS), min-heap (A*/Greedy). States move from frontier to visited upon expansion.

Special cases: IDA* uses path-based cycle detection (checking current path stack instead of global visited set). Simulated Annealing uses no visited set since it is a local search; stochastic acceptance provides the mechanism to escape local optima.

1.4 Discussion Questions (5 x 2 = 10 points)

a. Are h1 and h2 admissible? What if inconsistent or not admissible?

Both h1 and h2 are admissible (never overestimate true cost). h1 (Misplaced): each misplaced tile needs at least 1 move, so the count is a lower bound. h2 (Manhattan): each tile must travel at least its Manhattan distance; the sum never overestimates since it ignores tile interactions. **Both are also consistent:** for h2, moving one tile changes Manhattan distance by exactly +/-1, and step cost is 1, so h(n) <= c(n,a,n') + h(n'). If **not admissible**, A* may return suboptimal solutions. If **inconsistent** (but admissible), A* remains optimal but may re-expand states, hurting efficiency.

b. Compare A and IDA* time and space.*

Metric	A*	IDA*
Space	O(b^d) - all states in memory	O(d) - only current path
Time	O(b^d) - each state expanded once	O(b^d) - re-expands across iterations
Optimality	Optimal (admissible h)	Optimal (admissible h)
States Explored	4	8 (some re-exploration)
Path Cost	3 (optimal)	3 (optimal)

A* is faster per node but uses exponential memory. IDA* trades time (re-exploration) for O(d) space. For the 15-puzzle, A* runs out of memory; IDA* is preferred.

c. How did DFS prevent infinite paths?

Two mechanisms: (1) **Depth limit (max_depth=50):** prevents going deeper than 50 moves, guaranteeing termination. (2) **Global visited set:** checks `if neighbor not in visited` before adding to stack, preventing cycles (A->B->A->B...). The 8-puzzle has 181,440 reachable states, so the visited set ensures every state is explored at most once. Together, these guarantee DFS terminates.

d. BFS vs DFS memory. Why would one fail on 4x4?

BFS: O(b^d) -- stores entire frontier (all states at current depth) plus visited set. **DFS:** O(b*m) for the stack, but with visited set can use O(N). For the 3x3 puzzle both are manageable. **For 4x4 (15-puzzle):** The state space has ~10^{13} states, optimal solutions need 80+ moves. BFS would need to store b^{80} states in memory -- completely infeasible. DFS/IDA* with O(d) memory is required.

e. Minimax vs Alpha-Beta time and space.

Metric	Minimax	Alpha-Beta

Time Complexity	O(b^d)	O(b^(d/2)) best, O(b^(3d/4)) avg
Space Complexity	O(b*d)	O(b*d) (same)
Nodes Evaluated	766	546 (28.7% fewer)
Pruned Branches	0	141 (irrelevant nodes skipped)
Time	0.0012s	0.0009s (1.33x faster)
Optimality	Optimal	Optimal (same result)

Alpha-Beta prunes branches that cannot affect the final decision. In the best case, it effectively doubles the searchable depth for the same computation. Space is identical since both use DFS-style traversal.

1.5 Input / Output Format

Input (input.txt):

```
Start: 123;B46;758
Goal: 123;456;78B
```

Output (per algorithm):

```
Algorithm: [Name] ([Heuristic/Parameters])
Result: SUCCESS / FAILURE
Heuristic/Parameters: [e.g., h1, h2, T_init, cooling_rate, depth]
(Sub)Optimal Path: [Step-by-step actions]
Total States Explored: [count]
Total Time Taken: [seconds]
```

Summary of All Results:

Algorithm	Result	Path Len	States	Time (s)
BFS	SUCCESS	3	8	0.000034
DFS (depth=50)	SUCCESS	11	107,110	0.193252
Greedy (h1)	SUCCESS	3	4	0.000044
Greedy (h2)	SUCCESS	3	4	0.000032
A* (h1)	SUCCESS	3	4	0.000023
A* (h2)	SUCCESS	3	4	0.000030
IDA* (h1)	SUCCESS	3	8	0.000046
IDA* (h2)	SUCCESS	3	8	0.000032
Sim. Annealing	SUCCESS	Varies	17,397	0.047000
Minimax (d=6)	-	-	766	0.001200
Alpha-Beta (d=6)	-	-	546	0.000900

Question 2: Constraint Satisfaction Problem

Scenario: Assign 3 Security Bots (A, B, C) to 4 Time Slots to keep a digital library running 24/7.

2.1 Problem

Variables: {Slot1, Slot2, Slot3, Slot4}

Domains: Each slot assigned one bot from {A, B, C}

Constraints: (1) No Back-to-Back: a bot cannot work two consecutive slots. (2) Maintenance Break: Bot C cannot work in Slot 4. (3) Minimum Coverage: every bot must be used at least once.

2.2 Implementation (2 x 2.5 = 5 points)

a. Constraint Graph

The constraint graph has 4 nodes (time slots) connected by edges representing the No Back-to-Back constraint. An edge between Slot_i and Slot_{i+1} enforces that the assigned bots must differ.

```
Constraint Graph:
  [Slot1] ---!=--- [Slot2] ---!=--- [Slot3] ---!=--- [Slot4]
  {A,B,C}          {A,B,C}          {A,B,C}          {A,B}

  Edges (No Back-to-Back): (Slot1,Slot2), (Slot2,Slot3), (Slot3,Slot4)
  Unary Constraint: Slot4 domain reduced to {A,B} (C excluded)
```

b. First 3 Steps of Backtracking with MRV

Step 1: MRV selects **Slot4** (smallest domain: {A, B}, size=2). Assign Slot4=A. Forward Checking prunes A from Slot3's domain: Slot3={B, C}.

Step 2: MRV selects **Slot3** (domain {B, C}, size=2). Assign Slot3=B. Forward Checking prunes B from Slot2's domain: Slot2={A, C}.

Step 3: MRV selects **Slot2** (domain {A, C}, size=2). Assign Slot2=C. Forward Checking prunes C from Slot1's domain: Slot1={A, B}. All domains remain non-empty, so search continues.

Implementation:

```
class CSPSolver:
    def is_consistent(self, assignment, var, value):
        idx = self.slots.index(var)
        if idx &gt; 0:  # Check previous slot
            prev = self.slots[idx - 1]
            if prev in assignment and assignment[prev] == value: return False
        if idx &lt; len(self.slots) - 1:  # Check next slot
            nxt = self.slots[idx + 1]
            if nxt in assignment and assignment[nxt] == value: return False
        return True

    def mrv_select(self, assignment, domains):
        unassigned = [v for v in self.slots if v not in assignment]
        return min(unassigned, key=lambda v: len(domains[v]))

    def forward_check(self, assignment, domains, var, value):
        new_domains = {k: list(v) for k, v in domains.items()}
        idx = self.slots.index(var)
        for adj in [idx-1, idx+1]:
            if 0 &lt;= adj &lt; len(self.slots):
                slot = self.slots[adj]
                if slot not in assignment and value in new_domains[slot]:
                    new_domains[slot].remove(value)
                    if not new_domains[slot]: return None  # Domain wipeout
        return new_domains
    def backtrack(self, assignment, domains):
        if len(assignment) == len(self.slots):
            return assignment if self.check_minimum_coverage(assignment) else None
        var = self.mrv_select(assignment, domains)
        for value in domains[var]:
            if self.is_consistent(assignment, var, value):
                assignment[var] = value
                new_domains = self.forward_check(assignment, domains, var, value)
                if new_domains:
                    result = self.backtrack(assignment, new_domains)
                    if result: return result
                del assignment[var]
        return None
```

```
Result: SUCCESS | Heuristic: MRV | Inference: Forward Checking
Assignment: Slot1=C, Slot2=A, Slot3=B, Slot4=A
Total Assignments: 5 | Time: 0.000038 seconds
```

2.3 Discussion Questions (2 x 2.5 = 5 points)

a. Forward Checking failure detection

Suppose we assign Bot A to Slot 1 and Bot B to Slot 2. Forward Checking immediately prunes: A is removed from Slot 2's domain (already assigned) and from Slot 3's domain (no back-to-back with Slot 2). Similarly, B is removed from Slot 1 (already assigned) and Slot 3. Now suppose we have a very restricted domain, e.g., Slot 3 initially had only {A, B}. After pruning both A (back-to-back with Slot 2) and B (back-to-back with Slot 2), Slot 3's domain becomes empty {}. Forward Checking **immediately detects** this domain wipeout and triggers backtracking **before** attempting any assignment to Slot 3. Without Forward Checking, simple backtracking would try every value for Slot 3, fail on each, and only then backtrack -- wasting many more assignments.

b. Arc Consistency (AC-3) and the No Back-to-Back rule

This CSP is consistent. Arc consistency ensures that for every value in a variable's domain, there exists at least one consistent value in each neighboring variable's domain. AC-3 enforces this by maintaining a queue of arcs (directed edges). For each arc (Slot_i, Slot_j), it checks: for every value x in Slot_i's domain, does there exist a value y != x in Slot_j's domain? If not, x is pruned from Slot_i's domain.

In our problem, the initial domains are: Slot1={A,B,C}, Slot2={A,B,C}, Slot3={A,B,C}, Slot4={A,B} (after the unary constraint removes C). AC-3 processes arcs between consecutive slots. For each arc, since every domain has at least 2 values, every value in one domain has at least one different value available in the adjacent domain. For example, arc (Slot4, Slot3): for A in Slot4, we need y != A in Slot3 -- both B and C satisfy this. For B in Slot4, A and C in Slot3 satisfy it. No values are pruned, so the CSP is already arc-consistent after applying the unary constraint.

```
AC-3 Result: Arc Consistent = True
Reduced Domains: Slot1={A,B,C}, Slot2={A,B,C}, Slot3={A,B,C}, Slot4={A,B}
```

2.4 Input / Output Format

Input (input.txt):

```
Bots: A, B, C
Slots: Slot1, Slot2, Slot3, Slot4
Unary: Slot4 != C
```

Output:

```
Result: SUCCESS
Heuristic: MRV (Minimum Remaining Values)
Inference: Forward Checking
Constraints: No Back-to-Back, Maintenance Break (C!=Slot4), Min Coverage
Final Assignment: Slot1=C, Slot2=A, Slot3=B, Slot4=A
Total Assignments: 5
Total Time: 0.000038 seconds
```