University of Tehran

Faculty of Engineering

School of ECE

AI

CA1 - report sheet

[view project]

Niloufar Mortazavi

SID : 220701096

Question 1

1. State (Current Configuration)

Each state in the Lights Out puzzle represents the current configuration of the board, which is typically a 2D grid of lights. It’s like if every state is a screenshot of this moments board in the puzzle.

2. Initial State

The initial state is the configuration of the board at the beginning of the puzzle. This state is made by the random board function.

3. Goal State

The goal state is the configuration in which all the lights on the board are turned OFF. The objective of the puzzle is to reach this state by toggling the lights according to the rules. This state is identified by is solved function in the “LightsOutPuzzle” class when all the lights are OFF (all elements in puzzle equals to 0).

4. Actions

An action in the Lights Out puzzle consists of toggling a specific light on the board. However, toggling a light not only affects the light itself but also affects its neighboring lights (up, down, left, and right).

The toggling is done by the toggle function in the “LightsOutPuzzle” class also.

5. Solution

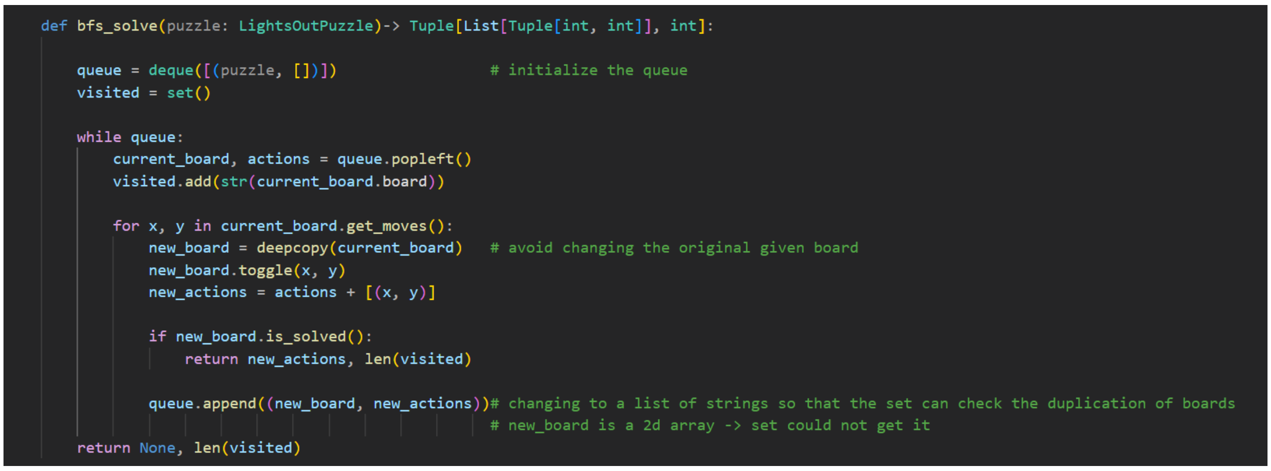
A solution to the Lights Out puzzle is a sequence of actions that transforms the initial state into the goal state. This can be represented as a list of moves that were made to toggle the lights in the correct order.

Question 2

1. Breadth-First Search (BFS)

How BFS Works:

- Breadth-First Search (BFS) is a graph traversal algorithm that explores all nodes at the current level before moving on to nodes at the next level.
- In the context of the Lights Out puzzle, BFS starts at the initial state and explores all possible configurations by toggling lights, level by level.



```python
def bfs_solve(puzzle: LightsOutPuzzle)-> Tuple[List[Tuple[int, int]], int]:

    queue = deque([(puzzle, [])])                 # initialize the queue
    visited = set()

    while queue:
        current_board, actions = queue.popleft()
        visited.add(str(current_board.board))

        for x, y in current_board.get_moves():
            new_board = deepcopy(current_board)   # avoid changing the original given board
            new_board.toggle(x, y)
            new_actions = actions + [(x, y)]

            if new_board.is_solved():
                return new_actions, len(visited)

            queue.append((new_board, new_actions))# changing to a list of strings so that the set can check the duplication of boards
                                                  # new_board is a 2d array -> set could not get it
    return None, len(visited)
```

Advantages:

- Complete: BFS will always find a solution if one exists.
- Optimal: In an unweighted graph (like the Lights Out puzzle where each action has the same cost), BFS is guaranteed to find the shortest path (solution with the fewest moves).

Disadvantages:

- High Memory Usage: Since BFS explores all possible states at the current depth, it requires storing a large number of nodes in memory.
- Time Complexity: The time complexity can be high because BFS may explore many irrelevant states before reaching the goal.

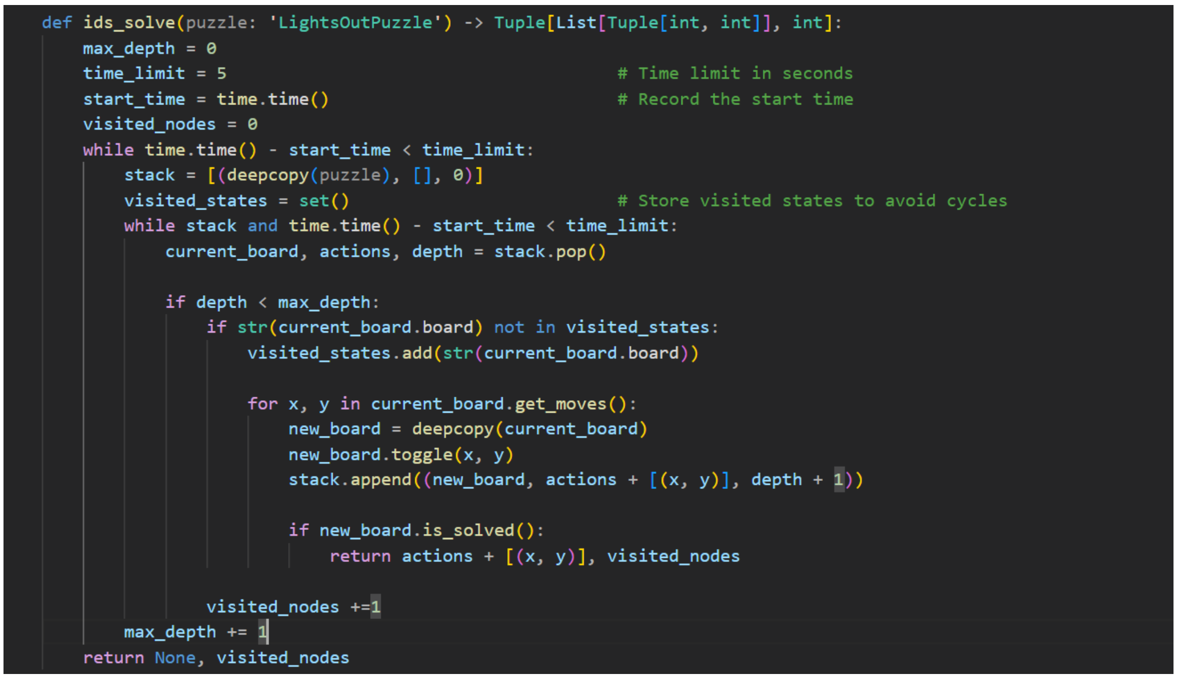
Is the Solution Optimal?

- Yes, BFS produces an optimal solution (the one with the fewest moves), assuming all actions have the same cost.

2. Iterative Deepening Search (IDS)

How IDS Works:

- Iterative Deepening Search (IDS) is a combination of Depth-First Search (DFS) and Breadth-First Search (BFS). It repeatedly runs a depth-limited DFS, increasing the depth limit with each iteration. It performs DFS with a depth limit of 0, then 1, then 2, and so on, until it finds a solution.



```
def ids_solve(puzzle: 'LightsOutPuzzle') -> Tuple[List[Tuple[int, int]], int]:
    max_depth = 0
    time_limit = 5                                      # Time limit in seconds
    start_time = time.time()                            # Record the start time
    visited_nodes = 0
    while time.time() - start_time < time_limit:
        stack = [(deepcopy(puzzle), [], 0)]
        visited_states = set()                          # Store visited states to avoid cycles
        while stack and time.time() - start_time < time_limit:
            current_board, actions, depth = stack.pop()

            if depth < max_depth:
                if str(current_board.board) not in visited_states:
                    visited_states.add(str(current_board.board))

                    for x, y in current_board.get_moves():
                        new_board = deepcopy(current_board)
                        new_board.toggle(x, y)
                        stack.append((new_board, actions + [(x, y)], depth + 1))

                        if new_board.is_solved():
                            return actions + [(x, y)], visited_nodes

                visited_nodes +=1
        max_depth += 1
    return None, visited_nodes
```

Advantages:

- Low Memory Usage: IDS inherits the low memory usage of DFS. At any point, IDS only needs to store the nodes along the current path, making its memory requirements much lower than BFS.
- Complete: Like BFS, IDS is complete; it will find a solution if one exists.

Disadvantages:

- Repeated Work: IDS performs repeated work because it re-explores nodes from the root to the current depth limit multiple times.

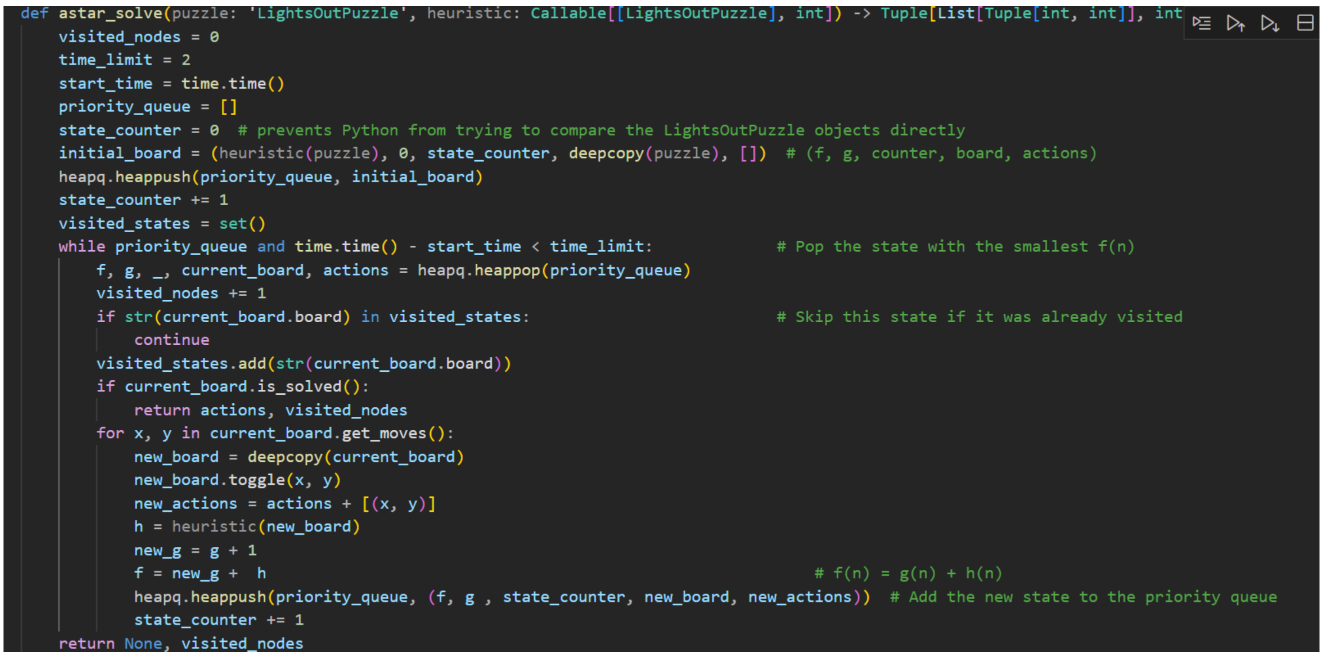
Is the Solution Optimal?

- Yes, IDS produces an optimal solution (the fewest moves) in scenarios where all actions have the same cost. Like BFS, it explores all nodes at the same depth before moving deeper, ensuring the first solution it finds is the optimal one.

3. A* Search

How A* Search Works:

- It uses a heuristic to guide its search toward the goal more efficiently.
- Used heapq to have a sorted list of frontier nodes where each state is prioritized based on a cost function: f(n) = g(n) + h(n)



```python
def astar_solve(puzzle: 'LightsOutPuzzle', heuristic: Callable[[LightsOutPuzzle], int]) -> Tuple[List[Tuple[int, int]], int
    visited_nodes = 0
    time_limit = 2
    start_time = time.time()
    priority_queue = []
    state_counter = 0  # prevents Python from trying to compare the LightsOutPuzzle objects directly
    initial_board = (heuristic(puzzle), 0, state_counter, deepcopy(puzzle), [])  # (f, g, counter, board, actions)
    heapq.heappush(priority_queue, initial_board)
    state_counter += 1
    visited_states = set()
    while priority_queue and time.time() - start_time < time_limit:             # Pop the state with the smallest f(n)
        f, g, _, current_board, actions = heapq.heappop(priority_queue)
        visited_nodes += 1
        if str(current_board.board) in visited_states:                          # Skip this state if it was already visited
            continue
        visited_states.add(str(current_board.board))
        if current_board.is_solved():
            return actions, visited_nodes
        for x, y in current_board.get_moves():
            new_board = deepcopy(current_board)
            new_board.toggle(x, y)
            new_actions = actions + [(x, y)]
            h = heuristic(new_board)
            new_g = g + 1
            f = new_g +  h                                                        # f(n) = g(n) + h(n)
            heapq.heappush(priority_queue, (f, g , state_counter, new_board, new_actions))  # Add the new state to the priority queue
            state_counter += 1
    return None, visited_nodes
```

Advantages:

- Optimal (if heuristic is admissible): A* guarantees finding the shortest path if the heuristic is admissible (never overestimates the actual cost).
- Efficient: A* is more efficient than BFS and DFS because the heuristic helps guide the search toward the goal, avoiding irrelevant paths.

Disadvantages:

- Memory Usage: A* requires more memory than DFS because it needs to maintain a priority queue of explored states.

Is the Solution Optimal?

- Yes, if the heuristic is admissible, A* produces an optimal solution (minimum number of moves).

4. Weighted A* Search

How Weighted A* Search Works:

- In WA*, the cost function is modified by multiplying the heuristic h(n) by a weight factor alpha. This allows the algorithm to behave more like a greedy search.

Advantages:

- Faster Search: By increasing the weight on the heuristic, WA* can explore fewer nodes, making the search faster in many cases.
- Flexibility: You can tune the weight w depending on the trade-off between speed and solution quality.

Question 3

1. Heuristic: Total number of disconnected groups of ON lights

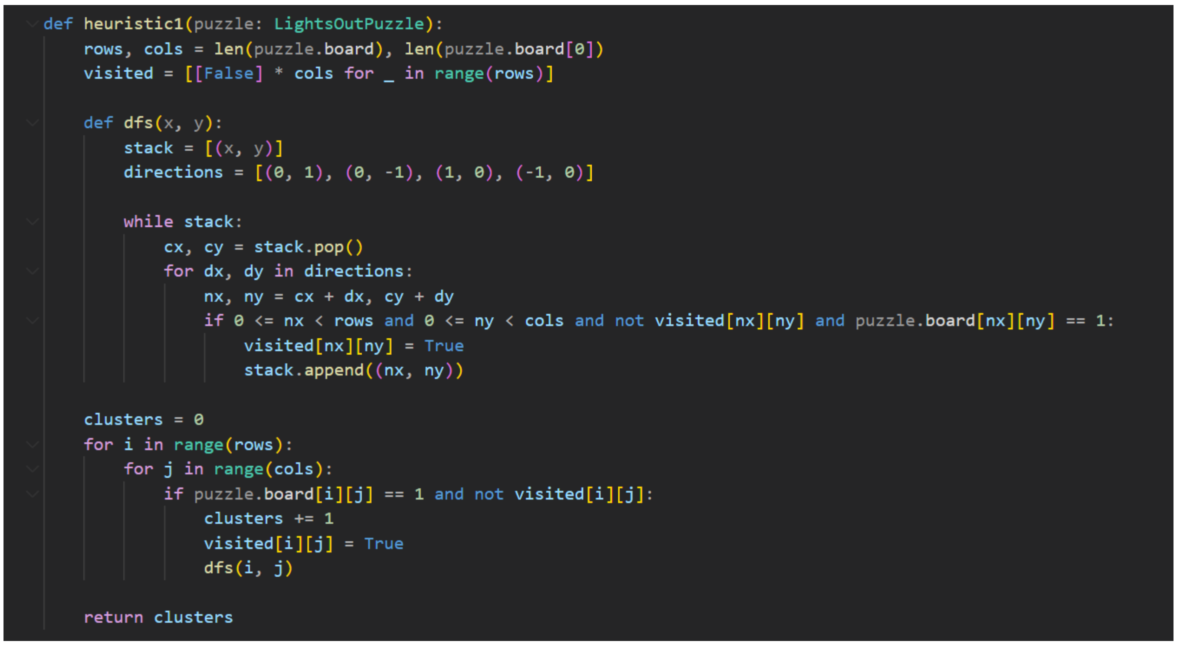
A "cluster" is a group of adjacent ON lights (connected vertically or horizontally). The idea is that each cluster will likely need at least one toggle to influence it, so more clusters suggest a higher number of moves may be needed to solve the puzzle.

Admissibility:

It is admissible. Each disconnected cluster is estimated to require at least one toggle to be affected. This is a safe lower bound because, in some cases, a single toggle can affect multiple clusters or more lights than anticipated. Therefore, the clusters heuristic cannot overestimate the actual number of moves required.

Consistency:

The clusters heuristic is consistent because every move reduces the number of clusters by at most one (or leaves it unchanged).



```python
def heuristic1(puzzle: LightsOutPuzzle):
    rows, cols = len(puzzle.board), len(puzzle.board[0])
    visited = [[False] * cols for _ in range(rows)]

    def dfs(x, y):
        stack = [(x, y)]
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        while stack:
            cx, cy = stack.pop()
            for dx, dy in directions:
                nx, ny = cx + dx, cy + dy
                if 0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny] and puzzle.board[nx][ny] == 1:
                    visited[nx][ny] = True
                    stack.append((nx, ny))

    clusters = 0
    for i in range(rows):
        for j in range(cols):
            if puzzle.board[i][j] == 1 and not visited[i][j]:
                clusters += 1
                visited[i][j] = True
                dfs(i, j)

    return clusters
```

2. Heuristic: Total number of lights that are ON

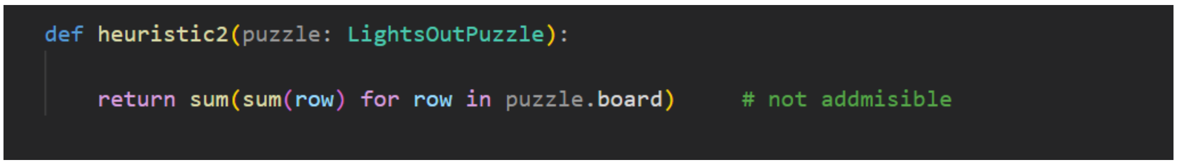
This heuristic counts the number of lights that are still ON. While it gives a rough idea of how far the current board is from the goal, it may overestimate the number of moves required because toggling one light can also turn off multiple neighboring lights.

Admissibility:

Admissible because the number of lights ON is a lower bound on the number of moves needed (in the worst case, each ON light might require a separate toggle).

Consistency:

Not Consistent because toggling a light can affect multiple other lights at once. For example, a single move may turn off several lights, resulting in a sudden drop in the heuristic value.



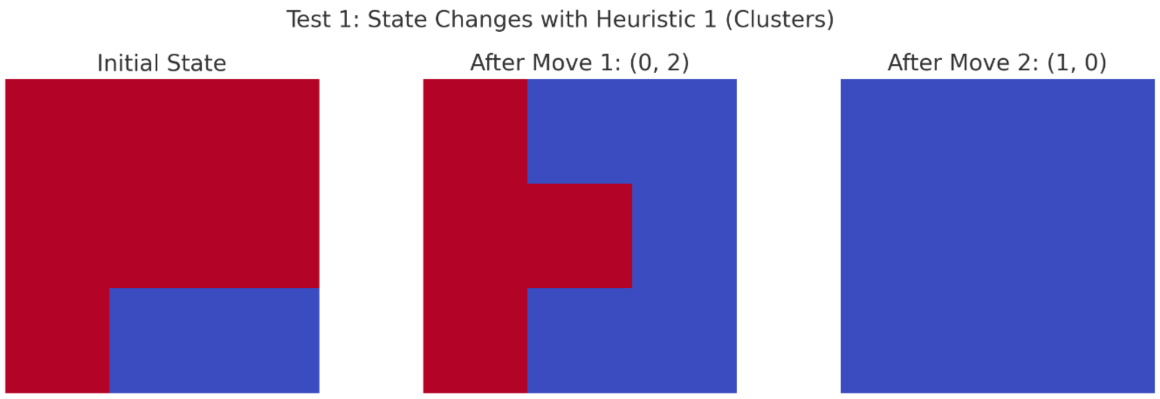
```python
def heuristic2(puzzle: LightsOutPuzzle):
    return sum(sum(row) for row in puzzle.board)     # not addmisible
```

Question 4

Running test:

[1 1 1]
[1 1 1]
[1 0 0]

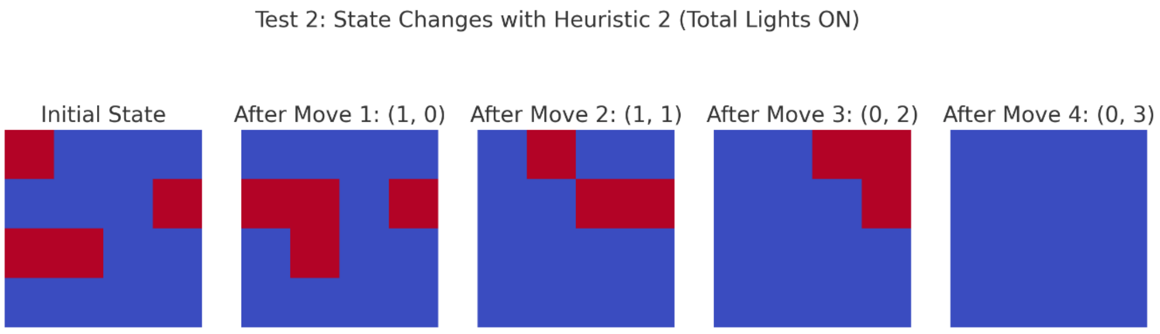
Heuristic	Solution	Nodes Visited
Heuristic 1	[(0, 2), (1, 0)]	7
Heuristic 2	[(1, 0), (0, 2)]	4



Running test:

[1 0 0 0]
[0 0 0 1]
[1 1 0 0]
[0 0 0 0]

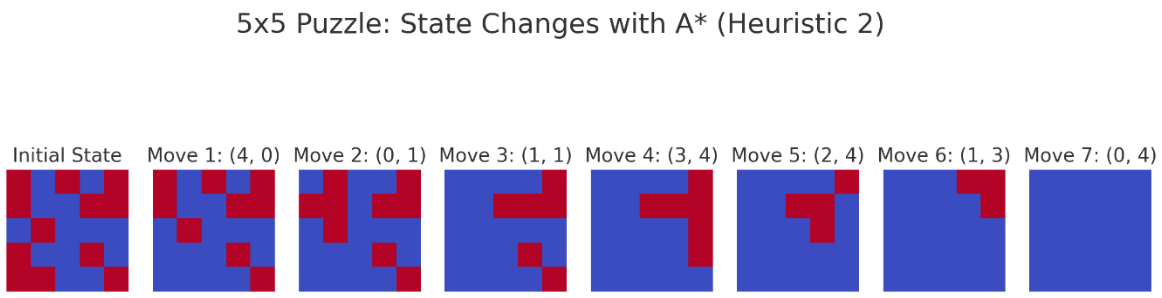
Heuristic	Solution	Nodes Visited
Heuristic 1	[(1, 1), (0, 3), (0, 2), (1, 0)]	375
Heuristic 2	[(1, 0), (1, 1), (0, 2), (0, 3)]	23



Running test:

```
[1 0 1 0 1]
[1 0 0 1 1]
[0 1 0 0 0]
[1 0 0 1 0]
[1 1 0 0 1]
```

Heuristic	**Solution**	**Nodes Visited**
Heuristic 1	Not found	-
Heuristic 2	[(4, 0), (0, 1), (1, 1), (3, 4), (2, 4), (1, 3), (0, 4)]	190



“Admissibility and Consistency Analysis”

Heuristic	**Admissible**	**Consistent**	**Explanation**
Heuristic 1 (Clusters)	Yes	Yes	It never overestimates the moves and decreases smoothly with each step.
Heuristic 2 (Total ON)	Yes	No	It can mislead the search since toggling one light may affect multiple lights, causing sudden drops in the heuristic value.

Why Does Heuristic 2 Perform Better Despite Inconsistency?

1. **Closer Alignment with the Goal State**:
   - Heuristic 2 (total lights ON) directly reflects how close the board is to the goal state (all lights OFF), even though it’s not perfectly consistent.
2. **Inconsistency Doesn’t Always Hurt**!
   - I always thought the consistent heuristic would be closer to reality!

     But inconsistency can cause backtracking in A*, As a result, Heuristic 2 still performs well and finds optimal solutions faster.
3. **Heuristic 1 Overestimates Complexity**:
   - Heuristic 1 (clusters) treats clusters as independent, which leads to more nodes being visited than necessary.

Question 5

View outputs table here: output data table.txt

Average Execution Time for Each Algorithm

Based on the test data, here are the average execution times for each algorithm:

- BFS: 0.529 seconds
- IDS: 1.640 seconds
- **A* (Heuristic 1 - Disconnected Clusters): 0.172 seconds
- **A* (Heuristic 2 - Total Lights ON): 0.292
- Weighted Heuristic 1 (alpha = 2): 0.061 seconds
- Weighted Heuristic 1 (alpha = 1000): 0.040 seconds
- Weighted Heuristic 2 (alpha = 2): 0.017 seconds
- Weighted Heuristic 2 (alpha = 1000): 0.003 seconds

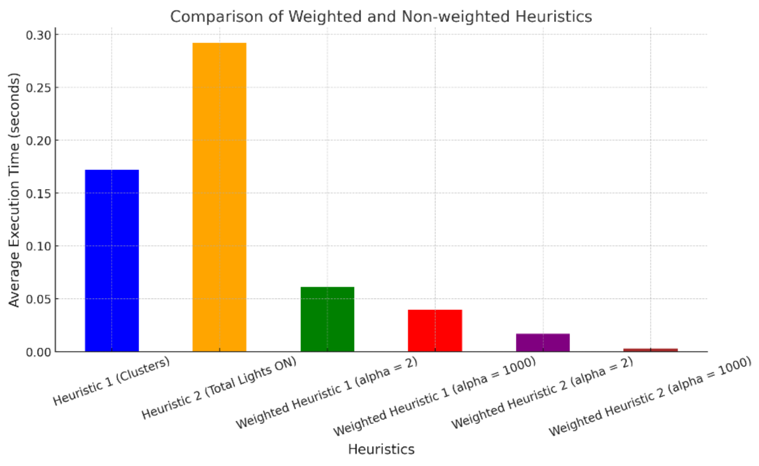
Analysis:

- BFS and IDS: These algorithms have the highest average execution times due to their exhaustive search nature, especially IDS, which explores nodes multiple times with increasing depth limits.

- A*: Both A* algorithms perform better than BFS and IDS. This is expected since heuristics guide the search more efficiently toward the solution.

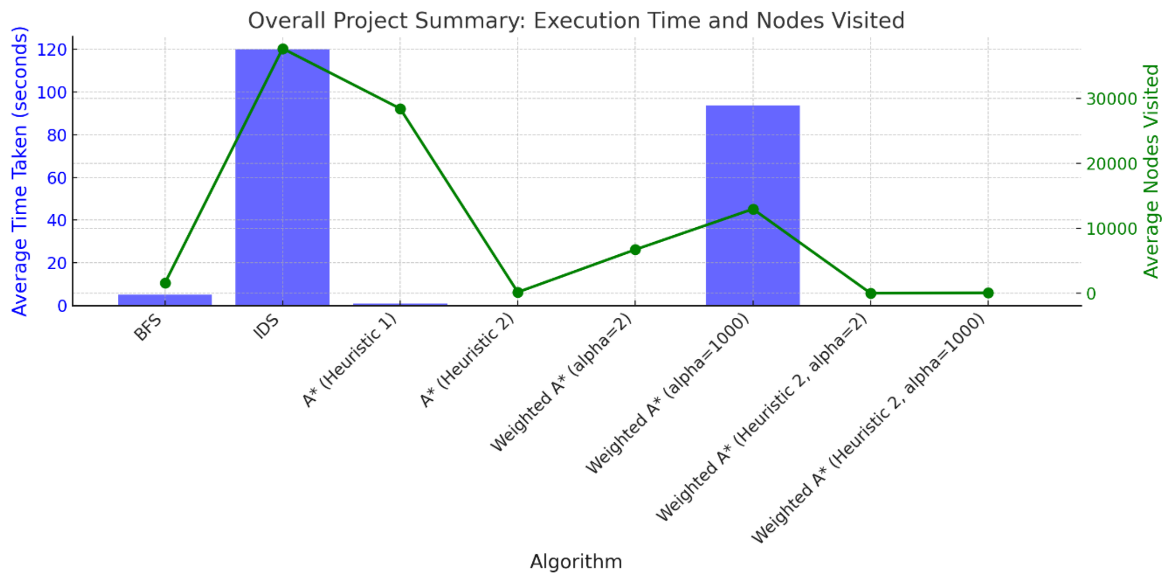
- A* with Heuristic 1 (Disconnected Clusters) performs slightly faster than Heuristic 2. However, the difference in average times is not large.

- Weighted Heuristic 2 (alpha = 1000) has the shortest execution time among all, showing that increasing the weight drastically reduces exploration time, but potentially at the cost of optimality.



Conclusion

BFS and IDS struggle due to exponential growth in the search space, leading to long execution times and excessive node exploration. In contrast, A* with well-chosen heuristics significantly reduces both execution time and nodes visited, but adding weights (as in weighted A* algorithms) further enhances performance, solving puzzles faster while exploring fewer nodes.


Overall Project Summary: Execution Time and Nodes Visited
Average Time Taken (seconds)
Average Nodes Visited
BFS
IDS
A* (Heuristic 1)
A* (Heuristic 2)
Weighted A* (alpha=2)
Weighted A* (alpha=1000)
Weighted A* (Heuristic 2, alpha=2)
Weighted A* (Heuristic 2, alpha=1000)
Algorithm