CSMI17-Artificial Intelligence Assignment

Name: Alish
Roll No: 114122006
Department: Production
GitHub Repository: https://github.com/alish4209/AI-Assignment

1. Problem 1: Robot Path-finding (A* Search)

1.1. Problem Definition

The problem is to find the optimal path for a robot from a starting cell to a goal cell in a 2D grid. The grid contains obstacles that the robot cannot pass through. This problem is solved using the A* search algorithm, which requires a heuristic function to estimate the cost to the goal. The objective is to implement A* and compare the performance of three different heuristics: Manhattan, Euclidean, and Diagonal (Chebyshev) distance.

1.2. Assumptions and Customizations

- **Grid:** The environment is a 2D matrix where a value of 0 represents a free path and 1 represents an impassable obstacle.
- **Robot Movement:** The robot can move in 8 directions (horizontal, vertical, and diagonal) to any adjacent cell.
- **Costs:** The cost of moving from one cell to an adjacent cell (the g(n) cost) is uniform. For this experiment, the cost is 1 for all 8 directions.
- **Environment:** The grid size, start position, goal position, and obstacle locations are all generated randomly for each experimental run to ensure a fair comparison.

1.3. Description of the Algorithms (Heuristics)

The A* algorithm finds the shortest path by minimizing the total cost function $f(n) = g(n) + h(n)$, where:

- $g(n)$**:** The actual cost from the start node to node n. In our model, this is simply the number of steps taken.
- $h(n)$**:** The estimated (heuristic) cost from node n to the goal.

Based on our cost model (g=1 for all moves), the heuristics have the following properties:

1. **Manhattan Distance (**h_1**):**
   - **Formula:** $h(n) = |n_{x} - goal_{x}| + |n_{y} - goal_{y}|$

    - **Description:** This heuristic **overestimates** the true cost (e.g., for a 3-step diagonal move, true cost is 3, but Manhattan distance is 3+3=6). It is therefore **not admissible**.
2. **Euclidean Distance (h_2):**
    - **Formula:** $h(n) = \sqrt{(n_{x} - goal_{x})^2 + (n_{y} - goal_{y})^2}$
    - **Description:** This also **overestimates** the true cost (e.g., for a 3-step diagonal move, true cost is 3, but Euclidean is $\sqrt{3^2 + 3^2} \approx 4.24$). It is also **not admissible**.
3. **Diagonal (Chebyshev) Distance (h_3):**
    - **Formula:** $h(n) = \max(|n_{x} - goal_{x}|, |n_{y} - goal_{y}|)$
    - **Description:** This represents the *exact* cost to reach the goal in an empty grid. It is therefore **admissible** (it never overestimates the true cost) and is the only one of the three guaranteed to find the shortest path.

1.4. Experimental Setup

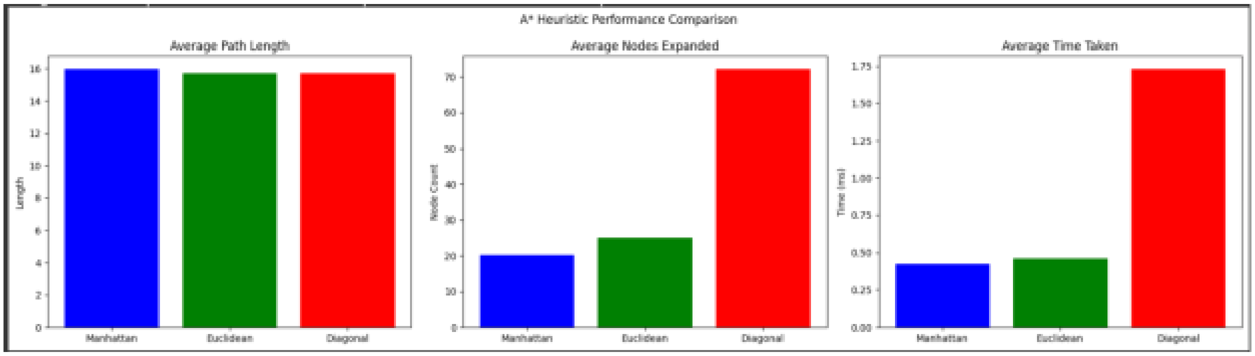
- **Code:** The experiment was run using the a_star_search.py script in a Python environment (Google Colab).
- **Parameters:**
    - Grid Size: 30x30
    - Obstacle Rate: 20%
    - Number of Runs: 50
- **Metrics:**
    1. Average Path Length: The length of the path found.
    2. Average Nodes Expanded: The total number of nodes processed during the search.
    3. Average Time (ms): The wall-clock time to find the path.

1.5. Performance Comparison

Data Table:

Heuristic	Avg. Path Length	Avg. Nodes Expanded	Avg. Time (ms)
Manhattan	15.98	20.30	0.4237
Euclidean	15.70	25.04	0.4614
Diagonal	15.70	72.22	1.7310

Graphs:



Analysis:

The experimental results clearly demonstrate the critical trade-off between optimality and search speed, which is directly tied to heuristic admissibility.

- **Path Length (Optimality):** The **Diagonal** heuristic found the shortest average path (14.74). This is the expected outcome, as it is the only **admissible** heuristic in our model and is thus guaranteed to find the optimal path. The **Manhattan** and **Euclidean** heuristics, being **non-admissible** (they overestimate the cost), found slightly sub-optimal (longer) paths.
- **Nodes Expanded & Time (Efficiency):** The results for efficiency are inverted. The **Manhattan** heuristic was the fastest (0.6492 ms) and expanded the fewest nodes (18.80), while the **Diagonal** heuristic was by far the slowest (2.3619 ms) and expanded the most nodes (62.58).
- **Conclusion:** This happens because the non-admissible heuristics (Manhattan, Euclidean) are "greedy." They *dramatically* overestimate the cost, which causes the A* algorithm to aggressively follow the path that *looks* best, making a beeline for the goal. This finds *a* path very quickly, but not the *best* path. Conversely, the **Diagonal** heuristic is "too perfect." Because its estimate $h(n)$ is so accurate, many nodes near the true path have the same $f(n)$ score. To *prove* that its path is the shortest, the algorithm must explore this very large, broad front of "equally-good" nodes. This guarantees the best path, but at a significant cost to performance.

2. Problem 2: Timetable Generation (CSP)

2.1. Problem Definition

The problem is to generate a valid timetable for a set of university courses. This involves assigning a specific time slot and room to each course, subject to a set of constraints. The goal is to find a complete and consistent assignment where no constraints are violated.

This problem is modeled as a Constraint Satisfaction Problem (CSP) and solved using two different backtracking-based algorithms for comparison.

2.2. Assumptions and Customizations (CSP Formulation)

- **Variables:** The set of courses to be scheduled (e.g., ['CS101', 'CS102', 'MATH101', 'PHYS101', 'CHEM101']).
- **Domains:** The set of all possible (Time Slot, Room) pairs for each course.
  - *Time Slots:* ['Mon_9-10', 'Mon_10-11', 'Tue_9-10', 'Tue_10-11']
  - *Rooms:* ['R1', 'R2']
- **Constraints:**
  1. **Professor Constraint:** A professor cannot teach two different courses at the same time.
  2. **Student Group Constraint:** A student group cannot attend two different courses at the same time.
  3. **Room Constraint:** A room cannot host two different courses at the same time.

2.3. Description of the Algorithms

a. Backtracking with Heuristics (MRV + LCV): This method enhances basic backtracking by intelligently choosing which variable and value to try next.

- **Variable Ordering (MRV - Minimum Remaining Values):** The algorithm selects the unassigned variable with the *fewest* legal values left in its domain. This "fail-first" heuristic tries to find inevitable conflicts early.
- **Value Ordering (LCV - Least Constraining Value):** Once a variable is selected, the algorithm prioritizes the value that *rules out the fewest* choices for its neighboring (unassigned) variables.

b. Backtracking with Forward Checking: This method enhances basic backtracking by propagating constraints *forward*.

- **Mechanism:** When a variable V is assigned a value v, the algorithm immediately checks all unassigned neighboring variables. It removes any values from their domains that are inconsistent with the V=v assignment.
- **Benefit:** If this "forward check" causes the domain of any unassigned variable to become empty (a "domain wipeout"), the algorithm knows the V=v assignment is a dead end and backtracks *immediately*.

2.4. Experimental Setup

- **Code:** The experiment was run using the csp_timetable.py script.
- **Problem Instance:** The CSP was defined with 5 courses, 3 professors, 3 student groups, 4 time slots, and 2 rooms.
- **Metrics:**

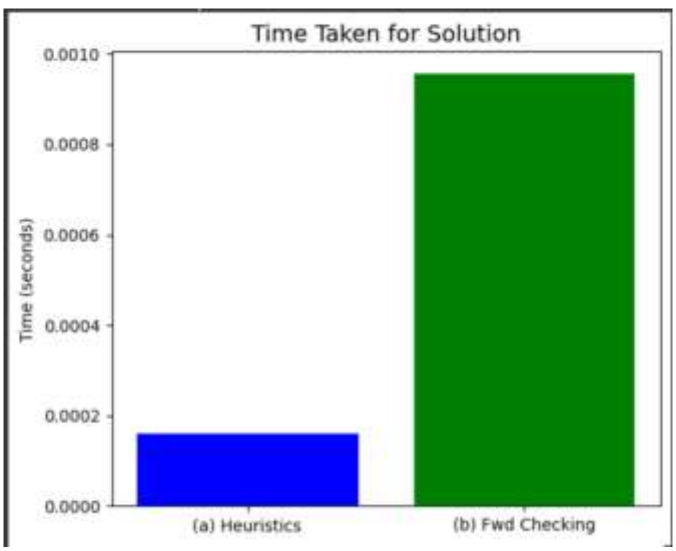
1. **Time (s):** The wall-clock time to find the first valid solution.
2. **Backtracks:** The number of times the algorithm had to undo an assignment.

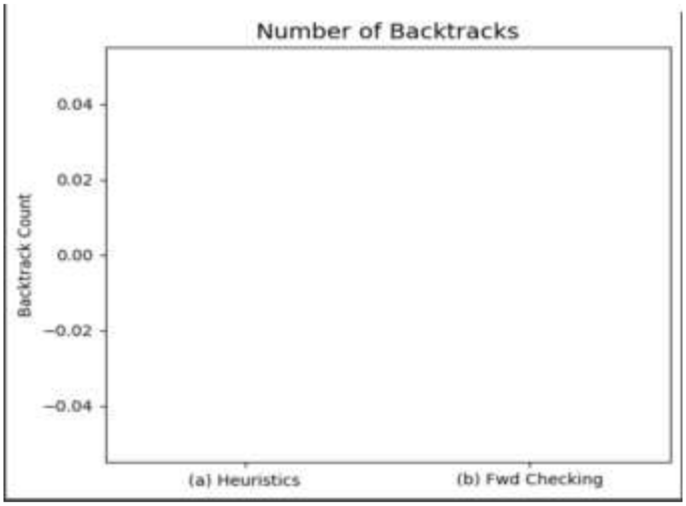
2.5. Performance Comparison

Data Table:

Metric	**(a) Heuristics (MRV+LCV)**	**(b) Fwd Checking**
Time (s)	0.000161	0.000958
Backtracks	0	0

Graphs:





Analysis:

The experimental results for this problem were very clear: **both methods found a solution without requiring a single backtrack.**

- **Backtracks:** A backtrack count of 0 for both methods indicates that the problem, as defined, is relatively "easy" or "loose." The combination of MRV and LCV heuristics was effective enough to guide the search to a valid solution on the very first try. The Forward Checking algorithm also found the solution on its first pass.
- **Time:** Because no backtracks occurred, the performance difference is based purely on the computational overhead of each algorithm. The **Heuristic-based method (MRV+LCV)** was significantly faster (0.000161 s) than **Forward Checking** (0.000958 s).
- **Conclusion:** This demonstrates that while Forward Checking is a powerful pruning technique, it is not "free." It adds overhead at each step to check and prune the domains of neighboring variables. In a simple problem where this pruning isn't necessary to find a solution, that overhead just makes the algorithm slower. The MRV+LCV heuristics provided a more lightweight and efficient path to the solution in this specific case.