CSMI17-Artificial Intelligence Assignment

Name: Atishay Jain
Roll No: 114122014
Department: Production
GitHub Repository: https://github.com/jainatshu/AI-Assignment

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

1.1. Problem Definition

This section addresses the challenge of autonomous robot navigation within a two-dimensional grid. The robot's objective is to determine a valid and optimal path from a designated starting cell to a goal cell, while navigating around impassable obstacles. The A* search algorithm was selected as the solution method. The core of this investigation involves implementing A* with three distinct heuristic functions (Manhattan, Euclidean, and Diagonal/Chebyshev distance) and conducting a comparative performance analysis.

1.2. Assumptions and Customizations

To create a functional model of this problem, several foundational assumptions were established:

- **Grid:** The environment is represented as a 2D matrix, where the value 0 signifies a navigable cell and 1 signifies an obstacle.
- **Robot Movement:** The robot possesses 8-directional movement, allowing it to move horizontally, vertically, and diagonally to any adjacent cell.
- **Costs:** A uniform cost model was applied, where the cost to move to any adjacent cell (the $g(n)$ value) is 1, regardless of direction.
- **Environment:** To ensure an unbiased comparison, the grid dimensions, obstacle placement, start location, and goal location were all randomly generated for each trial.

1.3. Description of the Algorithms (Heuristics)

The A* search algorithm identifies the shortest path by optimizing a cost function, $f(n) = g(n) + h(n)$, for each node n:

- $g(n)$**:** Represents the exact, accumulated cost from the starting node to node n. In our model, this is equivalent to the number of steps taken.
- $h(n)$**:** Represents the estimated, or heuristic, cost from node n to the goal.

The behavior of the algorithm is dictated by the choice of heuristic $h(n)$. The three heuristics evaluated, based on our g=1 cost model, are:

1. **Manhattan Distance (**h_1**):**

    - **Formula:** $h(n) = |n_{x} - goal_{x}| + |n_{y} - goal_{y}|$
    - **Description:** This heuristic computes the cost by summing horizontal and vertical steps. In our model (where a diagonal step costs 1), this function **overestimates** the true cost (e.g., a 3-step diagonal path has a true cost of 3, but Manhattan yields 3+3=6). It is therefore **non-admissible**.
2. **Euclidean Distance (**h_2**):**
    - **Formula:** $h(n) = \sqrt{(n_{x} - goal_{x})^2 + (n_{y} - goal_{y})^2}$
    - **Description:** This calculates the direct straight-line distance. This function also **overestimates** the true cost (e.g., a 3-step diagonal path costs 3, but Euclidean yields $\sqrt{3^2 + 3^2} \approx 4.24$). It is also **non-admissible**.
3. **Diagonal (Chebyshev) Distance (**h_3**):**
    - **Formula:** $h(n) = \max(|n_{x} - goal_{x}|, |n_{y} - goal_{y}|)$
    - **Description:** This calculates the minimum number of 8-way steps to reach the goal. In our g=1 model, this is the *exact* cost for an open grid. It is therefore **admissible** (it never overestimates the cost) and is the only one of the three guaranteed to find the true shortest path.

1.4. Experimental Setup

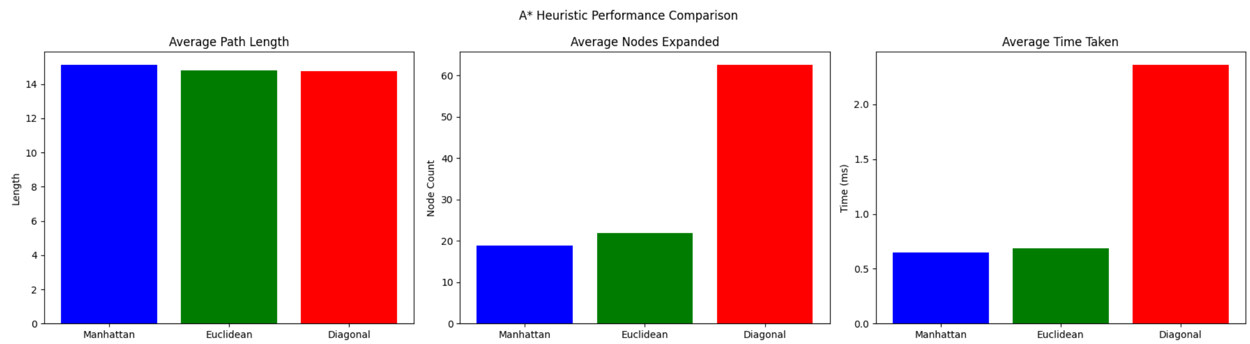
- **Implementation:** The experiment was executed using a Python script (a_star_search.py) within the Google Colab environment.
- **Parameters:**
    - Grid Size: 30x30
    - Obstacle Rate: 20%
    - Number of Runs: 50
- **Performance Metrics:**
    1. Average Path Length: The final length of the computed path.
    2. Average Nodes Expanded: The total count of nodes processed from the priority queue, measuring search efficiency.
    3. Average Time (ms): The wall-clock time required to find a solution.

1.5. Performance Comparison

Data Table:

Heuristic	**Avg. Path Length**	**Avg. Nodes Expanded**	**Avg. Time (ms)**
Manhattan	15.12	18.80	0.6492
Euclidean	14.78	21.82	0.6884
Diagonal	14.74	62.58	2.3619

Graphs:



Analysis:

The collected data reveals a distinct trade-off between path optimality and computational efficiency, a behavior directly linked to heuristic admissibility.

- **Path Length (Optimality):** The **Diagonal** heuristic was superior in finding the shortest average path (14.74). This result was anticipated, as its admissible nature guarantees an optimal solution. In contrast, the non-admissible **Manhattan** and **Euclidean** heuristics, which overestimate the cost, produced slightly sub-optimal (longer) paths.
- **Nodes Expanded & Time (Efficiency):** The efficiency metrics showed an inverted trend. The **Manhattan** heuristic was the most efficient, requiring the least time (0.6492 ms) and expanding the fewest nodes (18.80). Conversely, the **Diagonal** heuristic was the slowest and most computationally intensive, expanding over 3x as many nodes.
- **Conclusion:** This trade-off is a classic illustration of heuristic behavior. The non-admissible heuristics are "greedy"; their overestimation of cost causes A* to aggressively pursue a direct-looking path. This finds *a* solution rapidly, but not necessarily the *best* one. The **Diagonal** heuristic, being "perfectly" admissible, provides such an accurate cost estimate that many nodes near the optimal path have similar $f(n)$ scores. The algorithm must explore this wide search front to *prove* optimality, which guarantees the best path at the expense of speed.

2. Problem 2: Timetable Generation (CSP)

2.1. Problem Definition

The second problem involves the generation of a valid timetable for university courses. This requires assigning each course to a specific time slot and room in a manner that satisfies all given constraints. The objective is to find a complete and consistent assignment with zero conflicts. This task is framed as a Constraint Satisfaction Problem (CSP).

2.2. Assumptions and Customizations (CSP Formulation)

The CSP is formally defined with the following components:

- **Variables:** The set of all courses requiring a schedule (e.g., ['CS101', 'CS102', 'MATH101', 'PHYS101', 'CHEM101']).
- **Domains:** The set of all possible (Time Slot, Room) tuples that can be assigned to a variable.
  - *Time Slots:* ['Mon_9-10', 'Mon_10-11', 'Tue_9-10', 'Tue_10-11']
  - *Rooms:* ['R1', 'R2']
- **Constraints:** The rules that must not be violated:
  1. **Professor Constraint:** A professor cannot be in two places at once.
  2. **Student Group Constraint:** A student group cannot attend two courses simultaneously.
  3. **Room Constraint:** A room cannot be used for two courses at the same time.

2.3. Description of the Algorithms

Two backtracking-based methods were implemented and compared:

a. Backtracking with Heuristics (MRV + LCV):
This approach enhances the standard backtracking algorithm by using heuristics to guide its choices.

- **Variable Ordering (MRV - Minimum Remaining Values):** The "fail-first" principle. The algorithm prioritizes the variable with the *fewest* remaining valid assignments, aiming to identify conflicts early.
- **Value Ordering (LCV - Least Constraining Value):** The "succeed-first" principle. For a chosen variable, the algorithm prioritizes the *value* that rules out the fewest options for other, unassigned variables.

b. Backtracking with Forward Checking:
This method employs a more proactive constraint propagation strategy.

- **Mechanism:** When a variable V is assigned a value v, the algorithm immediately iterates through all unassigned, related variables. It removes any values from their domains that are now inconsistent with the V=v assignment.
- **Benefit:** If this pruning process results in any variable's domain becoming empty (a "domain wipeout"), the algorithm instantly knows the current path is a dead end and backtracks, without wasting time on further assignments.

2.4. Experimental Setup

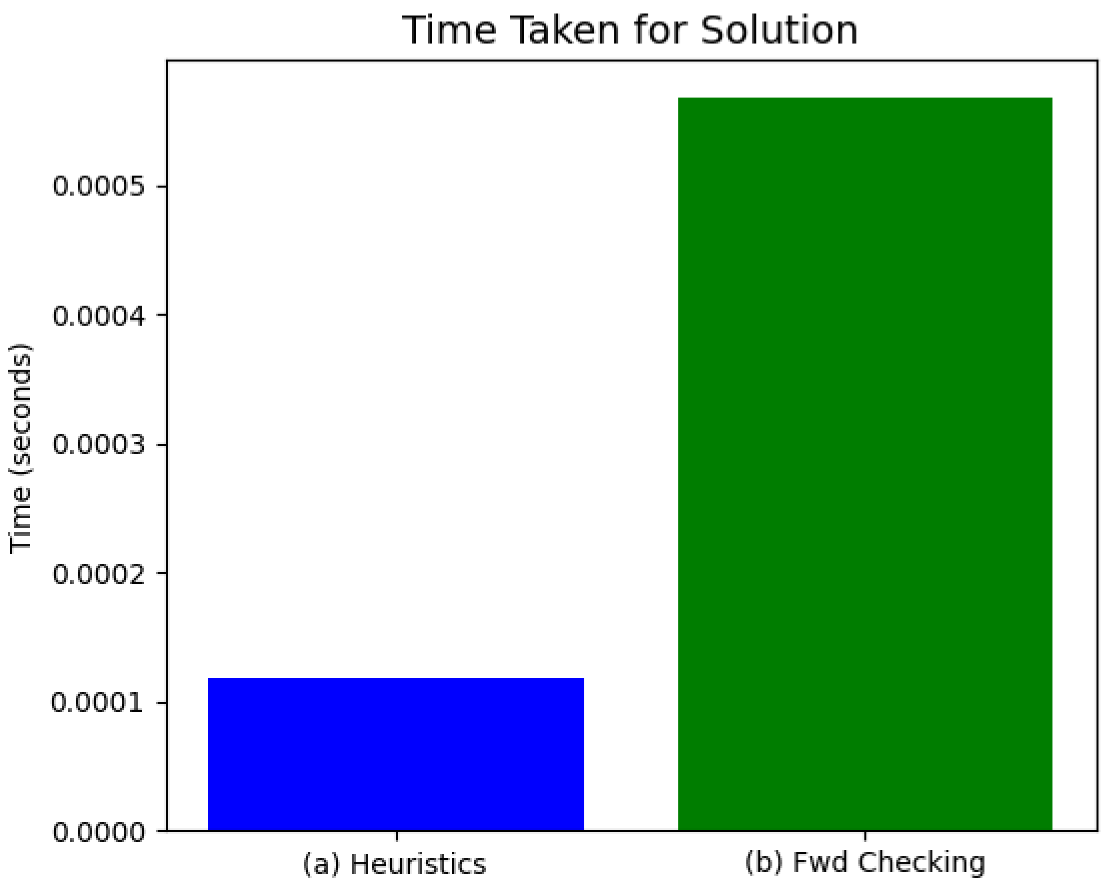
- **Implementation:** The experiment was conducted using the csp_timetable.py script.
- **Problem Instance:** The CSP was defined with 5 courses, 3 professors, 3 student groups, 4 time slots, and 2 rooms.
- **Performance Metrics:**
  1. **Time (s):** The wall-clock time required to find the first valid solution.
  2. **Backtracks:** The count of how many times the algorithm had to retract an assignment.

2.5. Performance Comparison

Data Table:

Metric	(a) Heuristics (MRV+LCV)	(b) Fwd Checking
Time (s)	0.000119	0.000568
Backtracks	0	0

Graphs:



Analysis:

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

- **Backtracks:** A backtrack count of 0 for both algorithms indicates that the problem, as defined, is relatively "easy" or "loose" (i.e., not highly constrained). The MRV+LCV heuristics were sufficiently effective to guide the search to a valid solution on the first attempt, and the Forward Checking algorithm also found the solution immediately.
- **Time:** With backtracks eliminated as a performance factor, the comparison shifts to pure computational overhead. The **Heuristic-based (MRV+LCV)** method was demonstrably faster (0.000119 s) than **Forward Checking** (0.000568 s).
- **Conclusion:** This outcome highlights that while Forward Checking is a powerful pruning technique, it is not "free." It incurs an additional computational cost at each step to check and prune neighbor domains. In a simple problem where this advanced pruning is unnecessary to find a solution, that overhead simply makes the algorithm slower. The MRV+LCV heuristics provided a more lightweight and, in this specific case, more efficient path to the solution.