Comparing Manhattan and Euclidean Heuristics in A* Algorithm Performance: Analyzing Runtime and Node Expansion Across Grid-Based Environments with Varying Obstacle Densities

Research Question: To what extent does the choice between the Manhattan distance and Euclidean distance as a heuristic function affect the runtime and number of nodes expanded by the A* algorithm in grid-based environments with varying obstacle densities?

Subject: Computer Science

Word Count: 3,993

Table of Contents

Introduction

Pathfinding algorithms are essential in computer science, enabling efficient navigation in applications like robotics, video games, and logistics. (Wikipedia contributors, Pathfinding) The A* algorithm is a prominent method, renowned for finding the shortest path in graph-based environments by using heuristic functions to guide its search. (GeeksforGeeks, A* Search Algorithm; Rubio) The choice of heuristic significantly impacts A*'s performance, balancing computational efficiency and path optimality. (GeeksforGeeks, A* Search Algorithm) This study compares two admissible heuristics—Manhattan distance, which sums horizontal and vertical differences (Chugani, Manhattan Distance), and Euclidean distance, which computes the straight-line distance via the Pythagorean theorem (Chugani, Euclidean Distance)—examining their effects on A*'s efficiency in grid-based environments.

To build on this foundation, the influence of heuristic choice becomes particularly evident in grid-based environments like mazes or room layouts, where obstacle density significantly alters pathfinding complexity. (Yap 2) For instance, robotics requires real-time navigation through cluttered spaces, and video games demand rapid pathfinding for smooth gameplay. Despite A*'s popularity, the performance of Manhattan and Euclidean heuristics under varying environmental constraints—especially runtime and nodes expanded—remains underexplored. This investigation seeks to fill this knowledge gap by evaluating these heuristics across grid-based settings with low, medium, and high obstacle densities.

To provide a structured approach to this investigation, subsets of the Moving AI Lab dataset (Sturtevant 144) were leveraged, testing mazes with variable corridor widths and room maps with random openings and differing individual room size, both of which on a 512×512 grid. This investigation excludes diagonal movement as it aligns with the Manhattan heuristic's axis-aligned nature, simplifying comparisons as a result. This extended essay thus investigates the extent to which the choice between Manhattan and Euclidean distance as a heuristic function affects A*'s runtime and number of nodes expanded, aiming to quantify differences and analyze their variation with obstacle density.

These insights pave the way for practical applications, where an efficient heuristic in autonomous navigation could reduce computational overhead, enhancing response times, and in video games, improved level design could minimize lag, enhancing player experience. This study thus offers actionable insights for developers and researchers using A* in grid-based applications, setting the stage for a deeper exploration.

Background Information

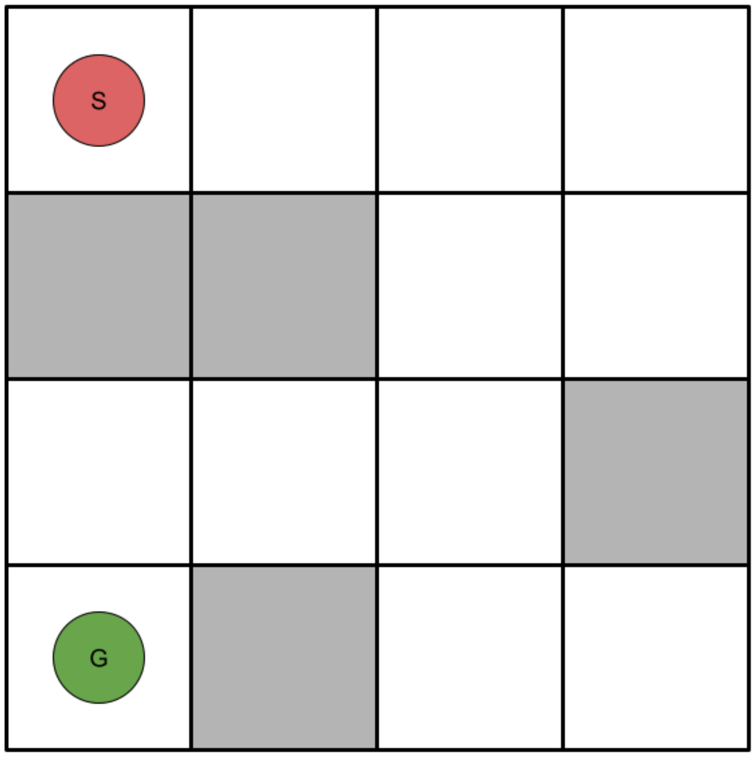
Building on the Introduction's context, this section provides the theoretical framework necessary to address the research question.

Foundational Concepts

Pathfinding is the process of determining a route from a starting point to a goal while avoiding obstacles (Wikipedia contributors, Pathfinding), a fundamental challenge in computer science with applications in Emergency Response Systems (e.g., disaster scenario terrain navigation), social networking (e.g., friend recommendations), and logistics (e.g., route optimization).(Memgraph)

This investigation works within **grid-based environments**, where a space is modeled as a grid of cells, similar to a chessboard, with each cell classified as passable or blocked by an obstacle. These grids provide a structured yet complex setting for pathfinding algorithms. (Sturtevant 144)

Figure 1 - A simple 4x4 grid with passable (white) and blocked (black) cells, showing a start (S) and goal (G).

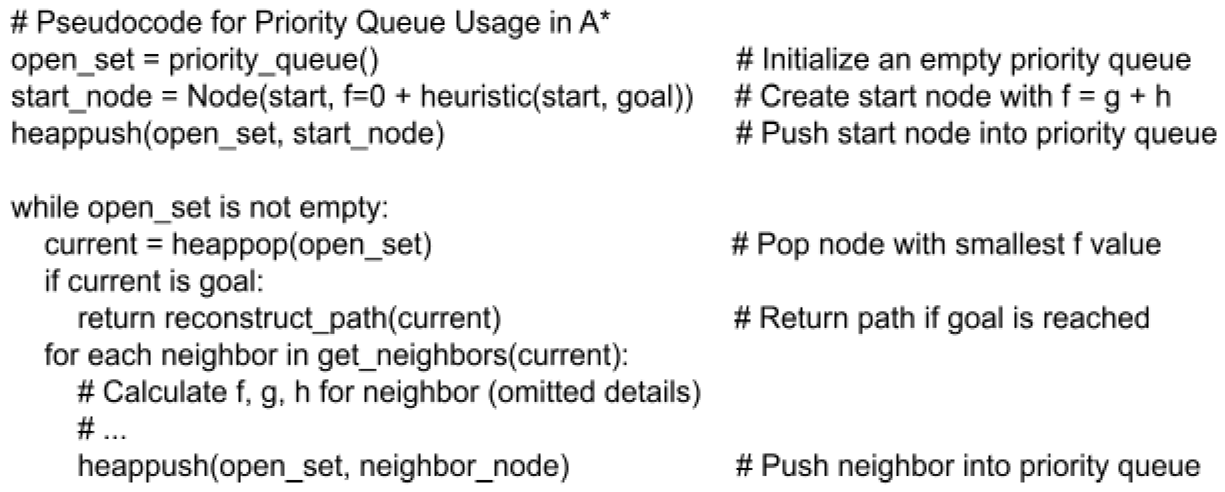


Algorithm and Core Mechanism

The $A*$ *algorithm* is a widely used pathfinding method that finds the shortest path efficiently by evaluating cells based on the actual cost from the start ($Gcost$) and an estimated cost to the goal ($Hcost$), summed as the total cost.($Fcost = Gcost - Hcost$). (GeeksforGeeks, A* Search Algorithm; Wikipedia contributors, A* Search Algorithm; Kumar; Lague) The **heuristic function** provides the h-cost, estimating the remaining distance to guide A* toward promising paths. (Patel, Heuristics) A good heuristic balances computational speed with accuracy. (Patel, Heuristics) Furthermore, a heuristic function is said to be **admissible** if it never overestimates the cost of achieving the goal. That is, the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path. (Wikipedia contributors, Admissible Heuristic)

A key enabler of A*'s efficiency is the **priority queue**, a data structure that organizes nodes by f-cost using a heap, ensuring the node with the lowest cost is processed next.(GeeksforGeeks, Priority Queue) This reduces the time to retrieve the next node from O(n) to O(log n), where n is the number of nodes, critical for large grids. (GeeksforGeeks, Priority Queue; Kumar)

Figure 2 - Pseudocode snippet for priority queue usage in A*:



```
# Pseudocode for Priority Queue Usage in A*
open_set = priority_queue()                              # Initialize an empty priority queue
start_node = Node(start, f=0 + heuristic(start, goal))   # Create start node with f = g + h
heappush(open_set, start_node)                           # Push start node into priority queue

while open_set is not empty:
    current = heappop(open_set)                          # Pop node with smallest f value
    if current is goal:
        return reconstruct_path(current)                 # Return path if goal is reached
    for each neighbor in get_neighbors(current):
        # Calculate f, g, h for neighbor (omitted details)
        # ...
        heappush(open_set, neighbor_node)                # Push neighbor into priority queue
```

The implementation of A* includes **tie-breaking** to resolve cases where nodes have equal $Fcosts$. Using a first-in, first-out (FIFO) approach, each node is assigned an order number based on when it was added. If $Fcosts$ are equal, the earlier node is chosen, ensuring consistent path selection.

Heuristic Functions

The investigation compares two heuristics: **Manhattan distance** and **Euclidean distance**.

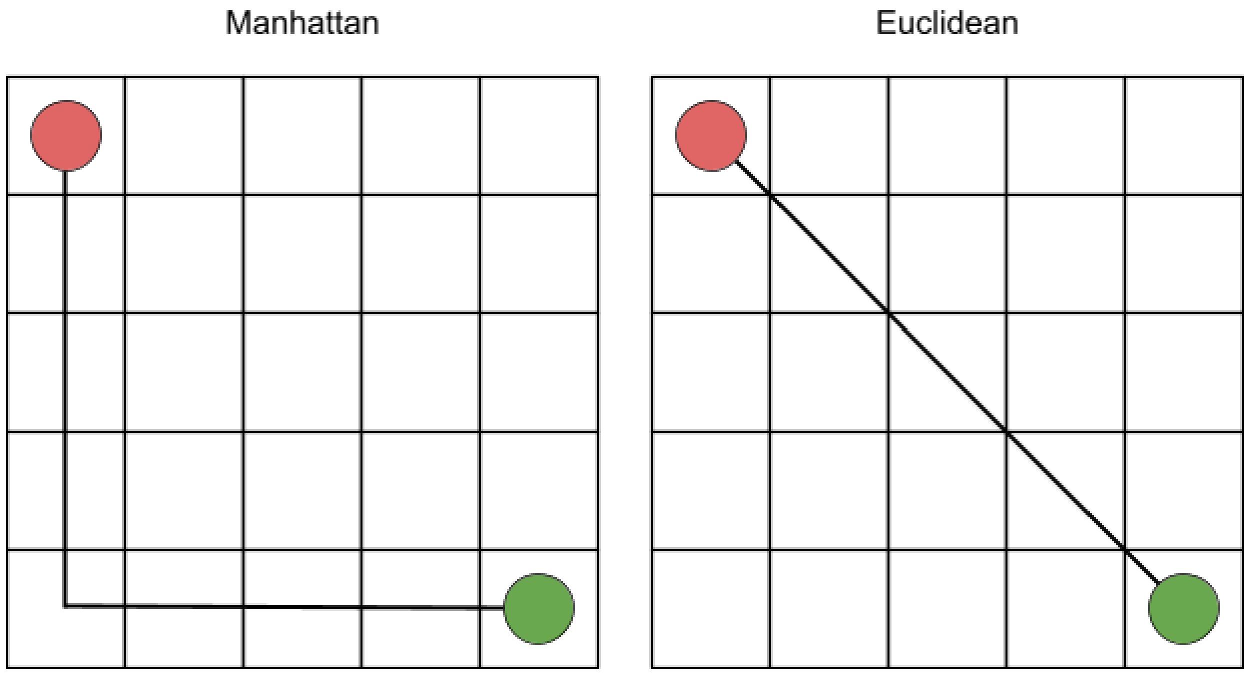
The **Manhattan distance** calculates the distance as the sum of absolute differences in the x and y coordinates, e.g., from $(1, 1)$ to $(3, 4)$, it is $|3 - 1| + |4 - 1| = 5$ steps, reflecting horizontal-vertical movement. This is computationally simple (no square roots) but less accurate for true shortest paths. (Chugani, Manhattan Distance; Kumar)

$h(n) = |x_1 - x_2| + |y_1 - y_2|$

The **Euclidean distance** measures the straight-line distance using the Pythagorean theorem: for $(1, 1)$ to $(3, 4)$, it is $\sqrt{(3 - 1)^2 + (4 - 1)^2} = \sqrt{13} \approx 3.61$units. This better approximates the true path but requires square root calculations, increasing cost. The choice of heuristic shapes A*'s exploration. (Chugani, Euclidean Distance)

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Figure 3 - Diagram comparing Manhattan and Euclidean paths.



Experimental Context

The investigation uses the **Moving AI dataset**, a standard collection of grid-based maps developed by Sturtevant, including mazes (winding paths) and rooms (open spaces with entry/exit points to other rooms). **Obstacle densities**, the proportion of blocked cells, are categorized in this investigation as low, medium, or high, where low density (few obstacles) eases navigation, and high density (many obstacles) increases complexity. Movement is restricted to **four-directional movement**—up, down, left, or right—excluding diagonals.

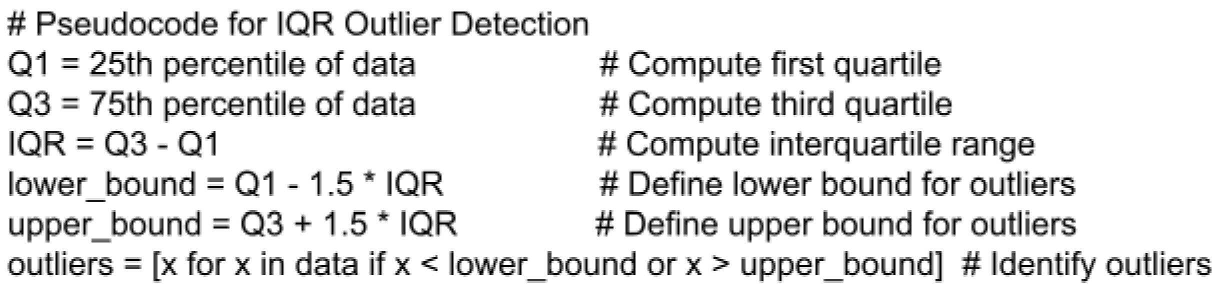
Performance Metrics

Runtime, measured in milliseconds (ms), is the time to compute a path, essential for real-time applications like gaming. **Nodes expanded** counts the cells A* explores, indicating search effort; more nodes suggest broader exploration, often linked to runtime. (Hart et al. 102)

Data Considerations and Analysis

Outliers, data points significantly deviating (e.g., high runtimes from complex paths), can be identified using the **Interquartile Range (IQR) method**. This calculates the range between the 25th percentile (Q_1) and 75th percentile (Q_3), with outliers lying beyond $Q_1 - 1.5 \times IQR$ or $Q_3 + 1.5 \times IQR$. (Moore et al. 45)

Figure 4 - Pseudocode for IQR:



```
# Pseudocode for IQR Outlier Detection
Q1 = 25th percentile of data                 # Compute first quartile
Q3 = 75th percentile of data                 # Compute third quartile
IQR = Q3 - Q1                                # Compute interquartile range
lower_bound = Q1 - 1.5 * IQR                 # Define lower bound for outliers
upper_bound = Q3 + 1.5 * IQR                 # Define upper bound for outliers
outliers = [x for x in data if x < lower_bound or x > upper_bound]  # Identify outliers
```

Correlation, via Pearson's r (ranging from -1 to 1), measures the relationship between variables like runtime and nodes expanded, where 1 is perfect positive, -1 perfect negative, and 0 no correlation.(GeeksforGeeks, Pearson Correlation Coefficient; Moore)

The A* implementation (from the provided code) initializes a $CustonNode$ class with x, y coordinates, $Gcost$, $Hcost$, $Fcost$, parent, and an order for tie-breaking. The $CustomAStar$ class uses a priority queue ($heapq$) to manage the open set, expanding nodes based on f-cost, with tie-breaking via the $\_It\_$ method comparing order. The $find\_path$ method tracks nodes_expanded, using $manhattan\_distance$ or $euclidean\_distance$ based on $heuristic\_type$.

Statistical analysis in the experiment averages runtime and nodes over 10 repetitions of the experiment, storing results in a CSV. The IQR method filters outliers, and Pearson's r could be computed post-analysis to correlate metrics.

Table 1 - Sample CSV output snippet (refer to Code Output section for the full CSV data):

map	**bucket**	**scenario**	**start_x**	**start_y**	**heuristic**	**density**	**avg_runtime_ms**	**avg_nodes_expanded**
maze512-16-0.map	0	0	348	495	manhattan	0.061523438	0.006484985	1
maze512-16-0.map	0	0	348	495	euclidean	0.061523438	0.004076958	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

map	bucket	scenario	start_x	start_y	heuristic	density	avg_runtime_ms	avg_nodes_expanded
8room_002.map	5	52	454	482	manhattan	0.210422516	3.64010334	1500
8room_002.map	5	52	454	482	euclidean	0.210422516	5.642795563	1908

Hypothesis

The A* algorithm's performance, measured in terms of runtime and number of nodes expanded, is expected to vary depending on the choice of heuristic (Manhattan or Euclidean distance) and the obstacle density of the grid-based environment. Specifically, it is hypothesized that the Manhattan distance heuristic will lead to shorter runtimes and fewer nodes expanded compared to the Euclidean distance heuristic in environments with high obstacle density (e.g., narrow corridors in mazes or small rooms), due to its simpler integer-based calculations and alignment with the grid's axis-aligned movement constraints. ("A* With Manhattan Distance or Euclidean Distance for Maze Solving?") In contrast, in environments with low obstacle density (e.g., wide corridors or large rooms), where more direct paths are available, the Euclidean distance heuristic is expected to perform comparably to the Manhattan heuristic in terms of runtime and nodes expanded, as its more accurate straight-line estimation may offset the computational overhead of floating-point calculations. Across all densities, the Manhattan heuristic is predicted to maintain a slight edge in runtime efficiency due to its lower computational complexity, while the number of nodes expanded may show greater variability depending on how each heuristic guides the search in different map structures (mazes vs. rooms).

Methodology

Building on the hypothesis, a controlled experiment was designed to investigate the extent to which the choice between the Manhattan distance and Euclidean distance as a heuristic function affects the runtime and number of nodes expanded by the A* algorithm in grid-based environments with varying obstacle densities. The methodology encompasses the implementation of the A* algorithm, the selection of appropriate datasets, the identification of key variables, the procedures for data collection, preprocessing, and analysis. The experiment was conducted using Python due to its robust libraries for pathfinding and data processing, and the implementation was optimized to ensure reproducibility while minimizing hardware-related variability as much as possible.

Experimental Setup

The A* algorithm was implemented in Python using the $CustomAStar$ class, supporting both Manhattan and Euclidean distance heuristics. Movement was restricted to four directions (up, down, left, right) with no diagonal movement, aligning with the Manhattan heuristic's axis-aligned nature and simplifying the comparison between heuristics. The implementation utilized Python's $heapq$ module for an efficient priority queue ($open\_set$) to manage nodes during the search. A $CustomNode$ class stored node properties, including $Fcost$ (total cost), $Gcost$ (cost from start), and $Hcost$ (heuristic estimate) values. A First-In-First-Out (FIFO) tie-breaking strategy was implemented using an $order$ attribute to prioritize nodes based on insertion order when f values were equal, ensuring consistent behavior.

Datasets from the Moving AI Lab were used, specifically subsets of “Mazes with variable cooridor width”[sic] and “Room maps - square rooms with random openings between them”[sic]. These datasets provided standardized, diverse structural challenges, focusing analysis on heuristic performance rather than map-specific irregularities. Mazes and room maps were categorized into three obstacle density levels—low, medium, and high—based on structural properties: corridor widths of 32 units (low), 16 units (medium), and 8 units (high) for mazes, and room dimensions of 32×32 (low), 16×16 (medium), and 8×8 (high) for rooms.

Variables

The experiment manipulated two independent variables:

1. **Heuristic Function**: Either Manhattan distance (sum of absolute differences in x and y coordinates) or Euclidean distance (straight-line distance using the Pythagorean theorem) was used as the heuristic in A*.
2. **Obstacle Density**: Three levels—low, medium, and high—were tested, defined by corridor widths in mazes and room sizes in room maps, providing varying spatial complexities.

The dependent variables measured were:

1. **Runtime Efficiency**: The average time (in milliseconds) taken by A* to find the shortest path, computed over multiple repetitions.

2. **Number of Nodes Expanded**: The average number of nodes added to the priority queue during the search, reflecting the algorithm's search effort.

Controlled variables included:

- **Map Dimensions**: Fixed at 512×512 cells to isolate effects of heuristic choice and obstacle density.
- **Movement Constraints**: Diagonal movement was disallowed to align with the Manhattan heuristic and simplify comparisons.
- **Tie-Breaking Strategy**: FIFO ordering for nodes with equal $Fcost$ values ensures reproducibility.
- **Hardware Configuration**: The experiment was performed on a Desktop PC with monitored CPU temperature and minimal background processes to reduce thermal throttling and resource contention between the experimental program and any background processes in the system. Hardware included a 12th Gen Intel(R) Core(TM) i5-12500 CPU, with 8GB of Samsung DDR4 RAM.
- **Implementation Language**: Python was used consistently, leveraging libraries like $heapq$ for priority queues.

Dataset and Scenarios

The Moving AI dataset included two subsets that were used in this investigation: mazes with variable corridor widths and room maps with random openings. Each subset had three maps per density level, totaling nine maze maps (e.g., *maze512-32-0.map* to *maze512-8-2.map*) and nine room maps (e.g., *32room_000.map* to *8room_002.map*). Each map was paired with a *.scen* file containing more than 200 pre-defined start and goal pairs for standardized test scenarios. Scenarios were processed until 50 successful runs per heuristic were achieved per map, skipping invalid scenarios (e.g., start or goal on an obstacle) with terminal messages logged for transparency.

Obstacle density was calculated programmatically using the *calculate_obstacle_density* function, defined as the fraction of obstacle cells (0) to total cells in the grid. For example, wider corridor mazes (e.g., *maze512-32-0.map*) had lower densities (e.g., 0.03), while narrower ones (e.g., *mazes512-8-0.map*) had higher densities (e.g., 0.11), providing a quantitative basis for density categorization.

Procedure

The experimental procedure followed these steps:

1. **Map and Scenario Loading**:
   - Maps were loaded from *.map* files, converting @ to 0 (obstacles) and '**.**' to 1 (free spaces), standardized to 512×512.
   - Scenarios were loaded from *.scen* files, extracting start and goal coordinates for 50 scenarios per map.
2. **Experiment Execution**:
   - The *run_experiment* function iterated over scenarios, running A* with both heuristics.
   - Each scenario was executed 10 times per heuristic to account for runtime variability, recording average runtime and nodes expanded.
   - Invalid runs (e.g., unreachable goals) were skipped until 50 successful runs per heuristic were achieved or scenarios were exhausted.

3. **Data Collection**:
   - Results were logged to a *.csv* file with columns for map name, scenario details (bucket, scenario index, start coordinates), heuristic, obstacle density, average runtime (ms), average nodes expanded, and success status.
   - Terminal output provided progress updates and warnings for incomplete runs.
4. **Data Aggregation**:
   - Mean runtime and nodes expanded were calculated across 10 repetitions per scenario, then averaged over up to 50 successful scenarios per heuristic per map.
   - This resulted in 1800 data points $(2\ datasets \times 3\ density\ levels \times 3\ maps\ per\ level \times 50\ scenarios \times 2\ heuristics)$, providing a robust sample for analysis.
5. **Data Preprocessing**:
   - Outliers were identified using the Interquartile Range (IQR) method within groups (dataset type, density level, heuristic). Values beyond $1.5{\times}IQR$ from the 25th or 75th percentiles were flagged as outliers for runtime and nodes expanded, logged in *results_with_outliers.csv*.
   - Outliers were retained in the dataset but flagged to assess their impact on summary statistics.

Justification of Methodological Choices

Python was chosen for its accessibility and libraries (e.g., *heapq*), minimizing hardware-related variability. The Moving AI dataset ensured controlled variability, focusing on heuristic performance rather than map irregularities, additionally it is a benchmark due to its standardized 512×512 grids and extensive scenario sets, widely used to evaluate pathfinding algorithms. Obstacle density categorization enabled systematic testing across spatial complexities. Excluding diagonal movement aligned with Manhattan's assumptions and simplified comparisons, while the FIFO tie-breaking strategy ensured reproducibility. Averaging 10 repetitions per scenario reduced runtime fluctuations, and capping at 50 successful runs balanced feasibility and reliability. Hardware controls (e.g., minimal background processes) and a fixed 512×512 grid size standardized the experiment, isolating heuristic and density effects.

Data Presentation

The experiment, detailed in the Methodology, involved 18 map files from the Moving AI dataset (9 mazes and 9 rooms, each with low, medium, and high obstacle densities), generating 1800 data points ($18\, files \times 50\, scenarios \times 2\, heuristics$), where each data point represents the average over 10 repetitions per scenario-heuristic combination. To analyze performance trends effectively, these data points were grouped into 12 categories based on three variables: dataset type (mazes or rooms), obstacle density (low, medium, high), and heuristic (Manhattan or Euclidean). Each group aggregates 150 scenarios ($3\, maps \times 50\, scenarios$), allowing for a robust comparison of density effects and heuristic performance, as required by the research question. The data underwent preprocessing, including outlier detection via the Interquartile Range (IQR) method, with outliers retained but flagged to assess their influence.

First, summary statistics were computed to quantify the average performance of each heuristic across the 12 groups. Table 2 below lists the mean runtime and mean number of nodes expanded for each combination of dataset type, density level, and heuristic, offering a direct comparison of Manhattan and Euclidean performance. These means, derived from 150 scenarios per group, provide a reliable measure of typical performance, addressing the research question's focus on runtime and search effort (nodes expanded).

Table 2: Mean Runtime and Nodes Expanded by Dataset Type, Density Level, and Heuristic

Dataset Type	Density Level	Heuristic	Mean Runtime (ms)	MeanNodes Expanded
Mazes	High	Manhattan	15.99	6,960.53
Mazes	High	Euclidean	18.16	7,447.11
Mazes	Medium	Manhattan	21.04	8,449.99
Mazes	Medium	Euclidean	23.10	8,713.95
Mazes	Low	Manhattan	11.69	4,953.27
Mazes	Low	Euclidean	13.13	5,210.39
Rooms	High	Manhattan	0.16	70.57

Dataset Type	Density Level	Heuristic	Mean Runtime (ms)	MeanNodes Expanded
Rooms	High	Euclidean	0.21	87.96
Rooms	Medium	Manhattan	0.28	145.89
Rooms	Medium	Euclidean	0.37	178.89
Rooms	Low	Manhattan	0.32	130.56
Rooms	Low	Euclidean	0.39	158.85

To directly address the research question's inquiry into the extent of performance differences, percentage differences between the heuristics were calculated using the formula:

$$Manhattan - Euclidean = Euclidean \times 100$$

Where negative values indicate Manhattan's advantage. Table 3 presents these percentage differences for both runtime and nodes expanded, building on the raw means to quantify the relative impact of heuristic choice across different environments and densities.

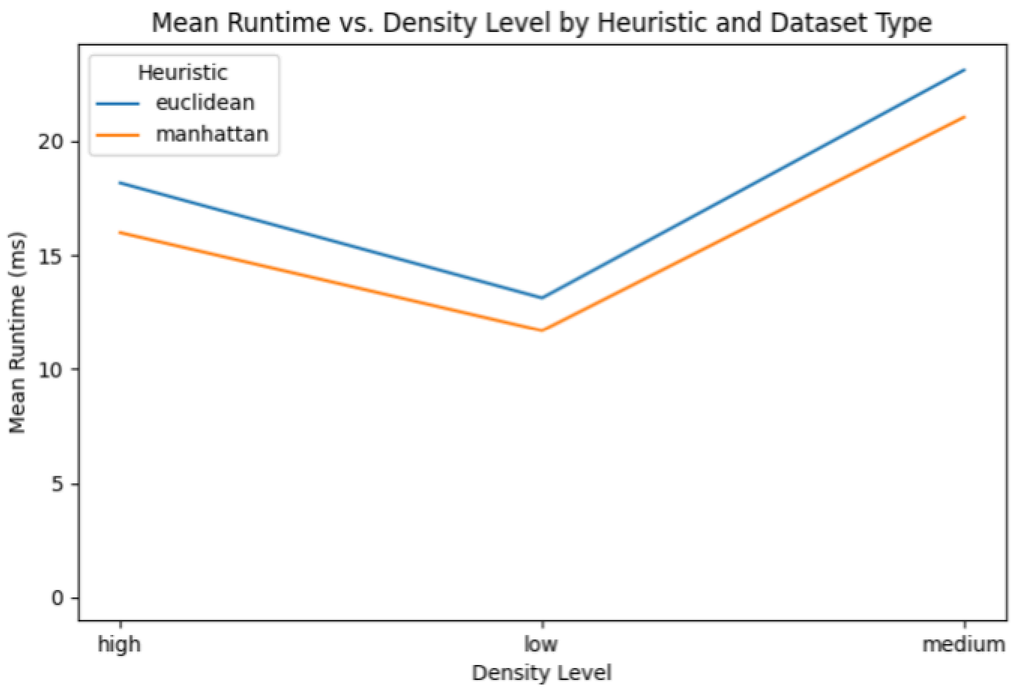
Table 2: Percentage Differences in Performance (Manhattan vs. Euclidean)

Dataset Type	Density Level	Runtime % Difference	Nodes Expanded % Difference
Mazes	High	-11.97%	-6.53%
Mazes	Medium	-8.92%	-3.03%

Dataset Type	**Density Level**	**Runtime % Difference**	**Nodes Expanded % Difference**
Mazes	Low	-10.96%	-4.93%
Rooms	High	-24.64%	-19.77%
Rooms	Medium	-23.23%	-18.44%
Rooms	Low	-17.85%	-17.81%

Next, to visually explore the trends suggested by these statistics, two figures were created to illustrate runtime patterns across density levels. Figure 5, a line graph, plots the mean runtime for each heuristic across low, medium, and high density levels, with dataset types distinguished by line style (solid for mazes, dashed for rooms). This visualization highlights how runtime varies with density and dataset type, revealing Manhattan’s consistent runtime advantage and the unexpected peak in mazes at medium density, which will be analyzed further in the next section.

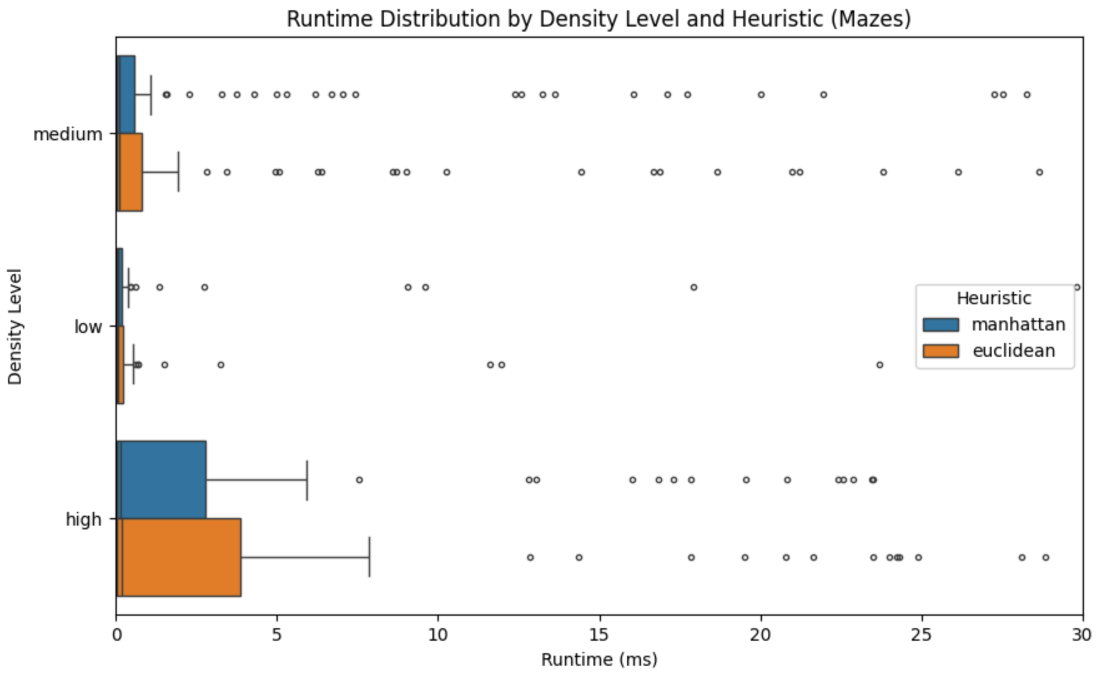
Figure 5: Mean Runtime vs. Density Level by Heuristic and Dataset Type



Caption: Line graph showing mean runtime (ms) for Manhattan and Euclidean heuristics across low, medium, and high density levels. Solid lines represent mazes, dashed lines represent rooms. Manhattan consistently exhibits lower runtimes, with larger differences in rooms. This graph was generated using

Building on the mean trends, Figure 6 provides a deeper look into runtime variability for mazes using a horizontal box plot. This plot displays the distribution of runtimes (medians, quartiles, and outliers) for each heuristic across density levels, capped at 30 ms to focus on typical values while noting extreme outliers (e.g., up to 661 ms) in the text. The graph shows the greater spread in medium-density mazes, which aligns with the higher means observed in Table 2.

Figure 6: Runtime Distribution by Density Level and Heuristic (Mazes)



Caption: Horizontal box plot of runtime distributions for mazes, displaying medians, quartiles, and outliers for each heuristic across density levels, capped at 30 ms to highlight typical runtimes. Medium-density mazes show the highest values; extreme outliers (e.g., up to 661 ms) are excluded from view but discussed in the text.

Finally, the impact of outliers on these results must be considered, as they were flagged during preprocessing. Outliers significantly inflated the mean runtimes, particularly in mazes, where excluding them reduced the mean in high-density mazes by up to 96% (e.g., from 18.16 ms to ~0.68 ms for Euclidean). However, the relative performance trends—such as Manhattan's advantage—remained consistent even after outlier removal, ensuring the reliability of the comparisons in Tables 2 and 3 and Figures 5 and

6. These outlier effects, along with the trends observed, will be interpreted in detail in the Data Analysis section to fully address the research question.

Data Analysis

This section analyzes the experimental results to evaluate the impact of heuristic choice (Manhattan vs. Euclidean distance) on the A* algorithm's performance, focusing on runtime and the number of nodes expanded across varying obstacle densities, as posed by the research question. The analysis draws on the summary statistics (Tables 2 and 3), visualizations (Figures 5 and 6), and outlier impacts presented earlier, interpreting key trends to determine the extent of performance differences and the role of obstacle density, while validating the hypothesis.

Manhattan consistently outperforms Euclidean across all environments, as shown in Table 2, with lower mean runtimes and fewer nodes expanded in every group. Table 3 quantifies this advantage: runtime reductions range from 8.92% to 11.97% in mazes and 17.85% to 24.64% in rooms, while nodes decrease by 3.03% to 6.53% in mazes and 17.81% to 19.77% in rooms. This supports the hypothesis, as Manhattan's simpler calculations (avoiding square roots) reduce computational overhead, a benefit amplified in rooms where larger differences (up to 24.64%) suggest its axis-aligned heuristic aligns better with their structure. The near-perfect correlation ($r = 0.998$) between runtime and nodes expanded reinforces this, showing Manhattan's reduced search effort (fewer nodes) directly drives its faster runtime. Figure 6 reveals that Euclidean's runtime distributions in mazes are slightly wider than Manhattan's, particularly in medium density, indicating Euclidean's greater sensitivity to path complexity, which may contribute to its larger node expansion (e.g., 7,447.11 vs. 6,960.53 in high-density mazes).

Obstacle density's effect reveals unexpected trends, challenging the hypothesis's focus on high-density environments. Figure 5 shows mazes peaking at medium density (21.04 ms Manhattan, 23.10 ms Euclidean) rather than high (15.99 ms Manhattan), with nodes following the same pattern (8,449.99 nodes Manhattan at medium density). This suggests medium-density mazes, balancing open and blocked areas, create complex paths that maximize search effort. Conversely, rooms show a decreasing trend from low (0.32 ms Manhattan, 130.56 nodes) to high density (0.16 ms, 70.57 nodes), likely due to smaller high-density rooms requiring less exploration. Figure 6 confirms medium-density mazes' higher variability, with the widest runtime distribution, reflecting inconsistent pathfinding challenges. The weak correlation ($r =- 0.079$) between density and runtime underscores that density's impact is map-specific, not universal.

Outliers inflate means (e.g., 96% reduction in high-density mazes when excluded), but trends hold, ensuring reliability. Manhattan's advantage is most pronounced in high-density rooms (24.64% runtime reduction), partially validating the hypothesis, though mazes show smaller differences. Thus, heuristic choice significantly affects A*'s efficiency, with density effects varying by map type, offering nuanced insights into pathfinding performance.

Limitations

While the experiment provides valuable insights into the performance of Manhattan and Euclidean heuristics in the A* algorithm, several limitations must be acknowledged to contextualize the findings. First, the study's scope is restricted to grid-based environments with four-directional movement (up, down, left, right), excluding diagonal movement, which is common in many real-world pathfinding applications such as primitive robotics, video games or industrial automatons.(Memgraph) This constraint may favor the Manhattan heuristic, which aligns with axis-aligned paths, potentially overestimating its advantage over Euclidean in scenarios allowing diagonal movement, where Euclidean might better approximate true distances.

Second, the presence of outliers significantly influenced the results (Sidhu), as noted in the Data Presentation section. In high-density mazes, excluding outliers reduced mean runtimes by up to 96% (e.g., from 18.16 ms to ~0.68 ms for Euclidean), indicating that a few extreme scenarios—possibly involving near-unpassable paths—skewed the averages. While trends remained consistent, this variability suggests the means may not fully reflect typical performance, particularly in mazes where runtime distributions were widest at medium density (Figure 6).

Third, the experiment used only two map types (mazes and rooms) with three density levels, which may not capture the full range of pathfinding challenges. For instance, environments with irregular obstacles or dynamic changes were not tested, limiting the generalizability of findings to broader contexts. Additionally, the sample size of 1800 data points, while robust, was averaged over 10 repetitions per scenario, which might not fully account for runtime variability due to system-specific factors like hardware performance.

Finally, the analysis focused solely on runtime and nodes expanded, omitting other metrics like path quality (e.g., path length or smoothness), which could provide a more comprehensive view of heuristic performance. These limitations suggest that while the findings are reliable within the experiment’s scope, they should be interpreted cautiously when applied to more complex pathfinding scenarios.

Further Development

The study’s findings suggest key areas for future research to address limitations and enhance A* heuristic understanding. Adapting the experiment to include eight-directional movement could highlight Euclidean’s potential advantage in approximating distances, possibly reducing Manhattan’s runtime edge (up to 24.64% in rooms). To mitigate outlier variability (e.g., 96% mean reduction in high-density mazes), a hybrid heuristic—switching between Manhattan and Euclidean based on local density—could be tested, potentially with machine learning for real-time optimization. Exploring diverse environments, such as urban grids with irregular obstacles or dynamic settings (e.g., traffic scenarios), could refine density impact analysis beyond the current three levels, addressing the weak correlation ($r =- 0.079$). Additionally, incorporating path quality metrics (e.g., length, smoothness) via a multi-objective optimization framework, tested in a simulated city, could offer a comprehensive view of heuristic efficiency, advancing A*’s practical applications.

Final Conclusion

This Extended Essay examined the impact of heuristic choice on A* algorithm performance, focusing on runtime and nodes expanded across grid-based environments with varying obstacle densities. Using 18 maps from the Moving AI dataset (9 mazes, 9 rooms), the experiment generated 1800 data points, grouped into 12 categories to compare Manhattan and Euclidean heuristics across low, medium, and high densities. Results from Tables 2 and 3 and Figures 5 and 6 show Manhattan outperforming Euclidean, with runtime reductions of 8.92% to 11.97% in mazes and 17.85% to 24.64% in rooms, and nodes expanded decreasing by 3.03% to 6.53% in mazes and 17.81% to 19.77% in rooms, supporting the hypothesis of Manhattan's efficiency advantage, especially in high-density rooms (24.64% difference).

However, obstacle density's effect varies, challenging the hypothesis's high-density focus. Mazes peaked at medium density (21.04 ms Manhattan) due to complex paths, while rooms decreased from low (0.32 ms) to high (0.16 ms) density, indicating map-specific influences. The weak correlation ($r = -0.079$) between density and runtime highlights environment-dependent performance, a key insight from the analysis.

Reflecting on the process, limitations like four-directional movement, limited map variety, and outlier-inflated means (though trends held) constrained generalizability, suggesting areas for improvement as noted in Further Development. Overall, Manhattan significantly boosts A* efficiency, particularly in rooms, answering the RQ with strong evidence of its superiority. This study validates the hypothesis and encourages further research into adaptive heuristics and diverse environments, enriching pathfinding algorithm development.

Bibliography

1. Wikipedia contributors. “Pathfinding.” *Wikipedia*, 17 Aug. 2024, en.wikipedia.org/wiki/Pathfinding. Accessed 18 Sept. 2024.
2. Kumar, Rajesh. *The A* Algorithm: A Complete Guide*. 7 Nov. 2024, www.datacamp.com/tutorial/a-star-algorithm. Accessed 15 Nov. 2024.
3. “A* With Manhattan Distance or Euclidean Distance for Maze Solving?” *Stack Overflow*, stackoverflow.com/questions/43906669/a-with-manhattan-distance-or-euclidean-distance-for-maze-solving. Accessed 18 Sept. 2024.
4. Wikipedia contributors. “A* Search Algorithm.” *Wikipedia*, 3 Sept. 2024, en.wikipedia.org/wiki/A*_search_algorithm. Accessed 18 Sept. 2024.
5. Sidhu, Harinder Kaur. Performance Evaluation of Pathfinding Algorithms. MS thesis. University of Windsor (Canada), 2020. https://scholar.uwindsor.ca/cgi/viewcontent.cgi?article=9230&context=etd.Accessed 18 Sept. 2024.
6. Rubio, Fatima. “Pathfinding Algorithms- Top 5 Most Powerful.” Graphable, 6 Jun. 2023, www.graphable.ai/blog/pathfinding-algorithms. Accessed 18 Sept. 2024.
7. Patel, Amit J. “*Introduction to the a* Algorithm*”. 2014, www.redblobgames.com/pathfinding/a-star/introduction.html. Accessed 18 Sept. 2024.
8. Monzonís Laparra, Daniel. "Pathfinding algorithms in graphs and applications." (2019). https://diposit.ub.edu/dspace/bitstream/2445/140466/1/memoria.pdf. Accessed 19 Sept. 2024.

9. Chugani, Vinod. “*What Is Manhattan Distance?”* 17 July 2024, www.datacamp.com/tutorial/manhattan-distance. Accessed 18 Sept. 2024.

10. Chugani, Vinod. “*Understanding Euclidean Distance: From Theory to Practice*.” 13 Sept. 2024, www.datacamp.com/tutorial/euclidean-distance. Accessed 19 Sept. 2024.

11. Sturtevant, Nathan R. "Benchmarks for Grid-Based Pathfinding." *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, 2012, pp. 144-148, webdocs.cs.ualberta.ca/~nathanst/papers/benchmarks.pdf. Accessed 19 Sept. 2024.

12. Hart, Peter E., et al. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, 1968, pp. 100-107. Accessed 23 Sept. 2024.

13. Moore, David S., et al. The Basic Practice of Statistics. 8th ed., W.H. Freeman, 2018. Accessed 14 Nov. 2024.

14. GeeksforGeeks. "Pearson Correlation Coefficient." *GeeksforGeeks*, www.geeksforgeeks.org/pearson-correlation-coefficient/. Accessed 14 Nov. 2024.

15. Wikipedia contributors. “Admissible Heuristic.” *Wikipedia*, 27 Nov. 2024, en.wikipedia.org/wiki/Admissible_heuristic. Accessed 18 Sept. 2024.

16. GeeksforGeeks. “A* Search Algorithm.” GeeksforGeeks, 30 Jul. 2024, www.geeksforgeeks.org/a-search-algorithm. Accessed 18 Sept. 2024.

17. Memgraph “Use-Cases of the Shortest Path Algorithm.” *Memgraph*, 3 Mar. 2022 memgraph.com/blog/use-cases-of-the-shortest-path-algorithm. Accessed 18 Sept. 2024.

18. GeeksforGeeks. “What Is Priority Queue | Introduction to Priority Queue.” *GeeksforGeeks*, 5 Aug. 2024, www.geeksforgeeks.org/priority-queue-set-1-introduction. Accessed 23 Sept. 2024.

19. Patel, Amit. "Heuristics". 2024, theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. Accessed 19 Sept. 2024.

20. Yap, Peter. "Grid-Based Path-Finding." *Conference of the Canadian Society for Computational Studies of Intelligence*, Springer, 2002, pp. 44-55, svn.sable.mcgill.ca/sable/courses/COMP763/oldpapers/yap-02-grid-based.pdf. Accessed 19 Sept. 2024.

21. Lague, Sebastian. “A* Pathfinding (E01: Algorithm Explanation).” *YouTube*, 16 Dec. 2014, www.youtube.com/watch?v=-L-WgKMFuhE. Accessed 19 Sept. 2024.

Appendix:

Body of Code

A* algorithm with Manhattan and Euclidean heuristics with tie breaking strategies and priority queue implementation.

~~~
```python
import heapq
import math
import time
from typing import List, Tuple, Dict, Set
import os
import csv
from collections import deque

# Custom Node class for A* with tie-breaking
class CustomNode:
    def __init__(self, x: int, y: int, g: float = 0, h: float = 0, parent=None, order: int = 0):
        self.x = x
        self.y = y
        self.g = g
        self.h = h
        self.f = g + h
        self.parent = parent
        self.order = order  # For FIFO tie-breaking

    def __lt__(self, other):
        if self.f != other.f:
            return self.f < other.f
        return self.order < other.order
~~~

```
# A* Implementation with custom heuristics and metrics tracking
class CustomAStar:
    def __init__(self, grid: List[List[int]], heuristic_type: str = "manhattan"):
        self.grid = grid
        self.height = len(grid)
        self.width = len(grid[0]) if self.height > 0 else 0
        self.heuristic_type = heuristic_type
        self.nodes_expanded = 0
        self.node_order_counter = 0

    def manhattan_distance(self, start: Tuple[int, int], goal: Tuple[int, int]) -> float:
        return abs(start[0] - goal[0]) + abs(start[1] - goal[1])

    def euclidean_distance(self, start: Tuple[int, int], goal: Tuple[int, int]) -> float:
        return math.sqrt((start[0] - goal[0]) ** 2 + (start[1] - goal[1]) ** 2)

    def get_heuristic(self, node: Tuple[int, int], goal: Tuple[int, int]) -> float:
        if self.heuristic_type == "manhattan":
            return self.manhattan_distance(node, goal)
        elif self.heuristic_type == "euclidean":
            return self.euclidean_distance(node, goal)
        else:
            raise ValueError("Unknown heuristic type")

    def get_neighbors(self, pos: Tuple[int, int]) -> List[Tuple[int, int]]:
        x, y = pos
        neighbors = []
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # No diagonal movement
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < self.height and 0 <= ny < self.width and self.grid[nx][ny] == 1:
                neighbors.append((nx, ny))
        return neighbors

    def find_path(self, start: Tuple[int, int], goal: Tuple[int, int]) -> Tuple[List[Tuple[int, int]],
int]:
        self.nodes_expanded = 0
        self.node_order_counter = 0

        open_set: List[CustomNode] = []
        closed_set: Set[Tuple[int, int]] = set()
        g_scores: Dict[Tuple[int, int], float] = {start: 0}
        came_from: Dict[Tuple[int, int], Tuple[int, int]] = {}

          start_node = CustomNode(start[0], start[1], 0, self.get_heuristic(start, goal), None,
self.node_order_counter)
        heapq.heappush(open_set, start_node)

        while open_set:
            current_node = heapq.heappop(open_set)
            current_pos = (current_node.x, current_node.y)

            if current_pos == goal:
                path = self.reconstruct_path(came_from, goal)
                return path, self.nodes_expanded

            closed_set.add(current_pos)
            self.nodes_expanded += 1

            for neighbor in self.get_neighbors(current_pos):
                if neighbor in closed_set:
```

```
                continue

            tentative_g = g_scores[current_pos] + 1

            if neighbor not in g_scores or tentative_g < g_scores[neighbor]:
                came_from[neighbor] = current_pos
                g_scores[neighbor] = tentative_g
                h_score = self.get_heuristic(neighbor, goal)
                self.node_order_counter += 1
                        neighbor_node = CustomNode(neighbor[0], neighbor[1], tentative_g,
h_score, current_pos, self.node_order_counter)
                heapq.heappush(open_set, neighbor_node)

        return [], self.nodes_expanded  # No path found

         def reconstruct_path(self, came_from: Dict[Tuple[int, int], Tuple[int, int]], goal:
Tuple[int, int]) -> List[Tuple[int, int]]:
        path = []
        current = goal
        while current in came_from:
            path.append(current)
            current = came_from[current]
        path.append(current)
        return path[::-1]

# Load a map file
def load_map(file_path: str) -> List[List[int]]:
    with open(file_path, 'r') as f:
        lines = f.readlines()

    # Skip header lines (type, height, width, map)
    grid = []
    expected_width = 512  # Since maps are supposed to be 512x512
    for line in lines[4:]:  # Map data starts after header
        row = []
        line = line.strip()
        for char in line:
            if char == '@':
                row.append(0)  # Obstacle
            elif char == '.':
                row.append(1)  # Free space
        # Pad or trim row to ensure consistent width
        if len(row) < expected_width:
            row.extend([0] * (expected_width - len(row)))  # Pad with obstacles
        elif len(row) > expected_width:
            row = row[:expected_width]  # Trim to expected width
        if row:  # Only append non-empty rows
            grid.append(row)

    # Ensure we have 512 rows
    height = len(grid)
    if height < 512:
        grid.extend([[0] * expected_width for _ in range(512 - height)])
    elif height > 512:
        grid = grid[:512]

    print(f"Loaded map {file_path}: {len(grid)}x{len(grid[0])}")
    return grid

# Load scenarios from a .scen file
def load_scenarios(file_path: str) -> List[Tuple[int, int, int, int, int]]:
    scenarios = []
    with open(file_path, 'r') as f:
        lines = f.readlines()

    # Skip header line ("version 1.0")
    for line in lines[1:]:
        parts = line.strip().split()
        if len(parts) >= 9:
            bucket = int(parts[0])  # First column is "Bucket"
            start_x = int(parts[4])
            start_y = int(parts[5])
            goal_x = int(parts[6])
            goal_y = int(parts[7])
            scenarios.append((bucket, start_x, start_y, goal_x, goal_y))
    return scenarios

# Calculate obstacle density
def calculate_obstacle_density(grid: List[List[int]]) -> float:
    total_cells = len(grid) * len(grid[0])
    obstacle_cells = sum(row.count(0) for row in grid)
    return obstacle_cells / total_cells

# Main experiment runner
def run_experiment():
    # Define map and scenario directories
    base_dir = "benchmark data"
    map_dirs = ["maze-map", "room-map"]
    scen_dirs = ["maze-scen", "room-scen"]

    # Results storage
    results = []
    target_successful_runs = 50  # We want 50 successful runs per map per heuristic
```

```
num_repetitions = 10

for map_dir, scen_dir in zip(map_dirs, scen_dirs):
    map_dir_path = os.path.join(base_dir, map_dir)
    scen_dir_path = os.path.join(base_dir, scen_dir)

    # Process each map file
    for map_file in sorted(os.listdir(map_dir_path)):
        if not map_file.endswith(".map"):
            continue

        map_path = os.path.join(map_dir_path, map_file)
        scen_file = map_file + ".scen"
        scen_path = os.path.join(scen_dir_path, scen_file)

        if not os.path.exists(scen_path):
            print(f"Scenario file not found for {map_file}")
            continue

        # Load map and scenarios
        try:
            grid = load_map(map_path)
        except Exception as e:
            print(f"Error loading map {map_file}: {e}")
            continue

        height = len(grid)
        width = len(grid[0])
        scenarios = load_scenarios(scen_path)
        density = calculate_obstacle_density(grid)
        print(f"Processing {map_file} with density {density:.2f}")

        # Process scenarios until we get 50 successful runs for each heuristic
        successful_runs = {"manhattan": 0, "euclidean": 0}
        scen_idx = 0

        while (scen_idx < len(scenarios) and
               (successful_runs["manhattan"] < target_successful_runs or
                successful_runs["euclidean"] < target_successful_runs)):
            bucket, start_x, start_y, goal_x, goal_y = scenarios[scen_idx]
            start = (start_x, start_y)
            goal = (goal_x, goal_y)

            # Check bounds before accessing grid
            if (start_x < 0 or start_x >= height or
                start_y < 0 or start_y >= width or
                goal_x < 0 or goal_x >= height or
                goal_y < 0 or goal_y >= width):
                  print(f"Skipping scenario {scen_idx} in {map_file}: Start ({start_x},{start_y})
or Goal ({goal_x},{goal_y}) out of bounds ({height}x{width})")
                scen_idx += 1
                continue

            # Check if start or goal is on an obstacle
            if grid[start_x][start_y] == 0 or grid[goal_x][goal_y] == 0:
                        print(f"Skipping scenario {scen_idx} in {map_file}: Start or Goal on
obstacle")
                scen_idx += 1
                continue

            # Run for both heuristics if needed
```

```
for heuristic in ["manhattan", "euclidean"]:
    if successful_runs[heuristic] >= target_successful_runs:
        continue

    # Initialize metrics
    total_runtime = 0
    total_nodes_expanded = 0
    success = True

    # Run repetitions
    for _ in range(num_repetitions):
        a_star = CustomAStar(grid, heuristic_type=heuristic)
        start_time = time.time()
        path, nodes_expanded = a_star.find_path(start, goal)
        end_time = time.time()

        runtime = (end_time - start_time) * 1000  # Convert to milliseconds
        total_runtime += runtime
        total_nodes_expanded += nodes_expanded

        if not path:  # No path found
            success = False
            break

    if success:
        # Compute averages
        avg_runtime = total_runtime / num_repetitions
        avg_nodes_expanded = total_nodes_expanded / num_repetitions

        # Increment successful runs for this heuristic
        successful_runs[heuristic] += 1
```

```
                        # Store result
                        results.append({
                            "map": map_file,
                            "bucket": bucket,
                            "scenario": scen_idx,
                            "start_x": start_x,
                            "start_y": start_y,
                            "heuristic": heuristic,
                            "density": density,
                            "avg_runtime_ms": avg_runtime,
                            "avg_nodes_expanded": avg_nodes_expanded,
                            "success": success
                        })
                    else:
                            print(f"Scenario {scen_idx} in {map_file} failed with {heuristic}: No path
found")

                scen_idx += 1

                                if    successful_runs["manhattan"]    <    target_successful_runs    or
successful_runs["euclidean"] < target_successful_runs:
                        print(f"Warning: Not enough valid scenarios for {map_file}. Manhattan:
{successful_runs['manhattan']}, Euclidean: {successful_runs['euclidean']}")
            else:
                        print(f"Completed {map_file}: {successful_runs['manhattan']} successful
Manhattan runs, {successful_runs['euclidean']} successful Euclidean runs")

    # Save results to CSV
    output_file = "results.csv"
    with open(output_file, 'w', newline='') as f:
        fieldnames = ["map", "bucket", "scenario", "start_x", "start_y", "heuristic", "density",
"avg_runtime_ms", "avg_nodes_expanded", "success"]
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writeheader()
        for result in results:
            writer.writerow(result)

    print(f"Results saved to {output_file}")

if __name__ == "__main__":
    run_experiment()
```

Statistical analysis programs

Outlier detection

```python
import pandas as pd
import numpy as np

# Load the dataset
df = pd.read_csv('results.csv')

# Ensure 'density' is numeric (if not already)
df['density'] = pd.to_numeric(df['density'], errors='coerce')

# Step 1: Define a function to detect outliers using the IQR method
def detect_outliers(group, column):
    Q1 = group[column].quantile(0.25)
    Q3 = group[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # Flag outliers
    outliers = (group[column] < lower_bound) | (group[column] > upper_bound)
    return outliers

# Step 2: Group data by dataset type, density level, and heuristic
# First, create a 'dataset_type' column to distinguish mazes and rooms
df['dataset_type'] = df['map'].apply(lambda x: 'mazes' if 'maze' in x.lower() else 'rooms')

# Create a 'density_level' column based on map names or density values
# Assuming map names indicate density (e.g., maze512-8-* for high, 32room_* for low)
def assign_density_level(map_name):
    if 'maze512-32' in map_name or '32room' in map_name:
        return 'low'
    elif 'maze512-16' in map_name or '16room' in map_name:
        return 'medium'
    elif 'maze512-8' in map_name or '8room' in map_name:
        return 'high'
    else:
        return 'unknown'

df['density_level'] = df['map'].apply(assign_density_level)

# Step 3: Apply outlier detection within each group for both metrics
df['runtime_outlier'] = False
df['nodes_outlier'] = False

# Group by dataset_type, density_level, and heuristic
grouped = df.groupby(['dataset_type', 'density_level', 'heuristic'])
```

```
# Detect outliers for runtime and nodes expanded
for name, group in grouped:
    # Detect outliers for avg_runtime_ms
    runtime_outliers = detect_outliers(group, 'avg_runtime_ms')
    df.loc[group.index, 'runtime_outlier'] = runtime_outliers

    # Detect outliers for avg_nodes_expanded
    nodes_outliers = detect_outliers(group, 'avg_nodes_expanded')
    df.loc[group.index, 'nodes_outlier'] = nodes_outliers

# Step 4: Summarize the number of outliers detected
print("Number of runtime outliers:", df['runtime_outlier'].sum())
print("Number of nodes expanded outliers:", df['nodes_outlier'].sum())

# Step 5: Optionally, inspect some outliers
runtime_outliers = df[df['runtime_outlier'] == True][['map', 'heuristic', 'density_level',
'avg_runtime_ms']]
nodes_outliers = df[df['nodes_outlier'] == True][['map', 'heuristic', 'density_level',
'avg_nodes_expanded']]
print("\nSample runtime outliers:\n", runtime_outliers.head())
print("\nSample nodes expanded outliers:\n", nodes_outliers.head())

# Step 6: Save the updated dataset with outlier flags
df.to_csv('results_with_outliers.csv', index=False)
print("Updated dataset saved as 'results_with_outliers.csv'")
```

Summary statistics

```python
import pandas as pd
from scipy.stats import pearsonr
```

```
import sys
from contextlib import contextmanager

# Create a custom context manager to handle output
@contextmanager
def output_to_file_and_console(filename):
    class FileAndConsole:
        def __init__(self, filename):
            self.terminal = sys.stdout
            self.file = open(filename, 'w')

        def write(self, message):
            self.terminal.write(message)
            self.file.write(message)

        def flush(self):
            self.terminal.flush()
            self.file.flush()

    original_stdout = sys.stdout
    sys.stdout = FileAndConsole(filename)
    try:
        yield
    finally:
        sys.stdout.file.close()
        sys.stdout = original_stdout

# Main analysis code wrapped in the context manager
with output_to_file_and_console('summary_stats.txt'):
    # Load the dataset with outlier flags
    df = pd.read_csv('results_with_outliers.csv')
```

```
# Ensure columns are numeric
df['avg_runtime_ms'] = pd.to_numeric(df['avg_runtime_ms'], errors='coerce')
df['avg_nodes_expanded'] = pd.to_numeric(df['avg_nodes_expanded'], errors='coerce')
df['density'] = pd.to_numeric(df['density'], errors='coerce')

# Step 1: Overall Summary Statistics
overall_summary = df.groupby('heuristic').agg({
    'avg_runtime_ms': ['mean', 'std', 'median', 'min', 'max'],
    'avg_nodes_expanded': ['mean', 'std', 'median', 'min', 'max']
}).reset_index()
print("Overall Summary Statistics:\n", overall_summary)

# Step 2: Summary Statistics by Grouping (Dataset Type, Density Level, Heuristic)
# Ensure dataset_type and density_level are defined (as in previous code)
df['dataset_type'] = df['map'].apply(lambda x: 'mazes' if 'maze' in x.lower() else 'rooms')
def assign_density_level(map_name):
    if 'maze512-32' in map_name or '32room' in map_name:
        return 'low'
    elif 'maze512-16' in map_name or '16room' in map_name:
        return 'medium'
    elif 'maze512-8' in map_name or '8room' in map_name:
        return 'high'
    else:
        return 'unknown'
df['density_level'] = df['map'].apply(assign_density_level)

grouped_summary = df.groupby(['dataset_type', 'density_level', 'heuristic']).agg({
    'avg_runtime_ms': ['mean', 'std', 'median'],
        'avg_nodes_expanded': ['mean', 'std', 'median'],
        'map': 'count'  # Count of scenarios
    }).reset_index()

    # Flatten column names
    grouped_summary.columns = ['dataset_type', 'density_level', 'heuristic',
                             'mean_runtime_ms', 'std_runtime_ms', 'median_runtime_ms',
                                               'mean_nodes_expanded',  'std_nodes_expanded',
'median_nodes_expanded',
                             'scenario_count']
    print("\nGrouped Summary Statistics:\n", grouped_summary)

    # Calculate percentage difference between heuristics within each group
    # Pivot to compare Manhattan vs. Euclidean
    pivot_runtime = grouped_summary.pivot_table(index=['dataset_type', 'density_level'],
                                          columns='heuristic',
                                          values='mean_runtime_ms').reset_index()
                          pivot_runtime['percent_diff']       =       ((pivot_runtime['manhattan']       -
pivot_runtime['euclidean']) / pivot_runtime['euclidean']) * 100

    pivot_nodes = grouped_summary.pivot_table(index=['dataset_type', 'density_level'],
                                        columns='heuristic',
                                        values='mean_nodes_expanded').reset_index()
      pivot_nodes['percent_diff'] = ((pivot_nodes['manhattan'] - pivot_nodes['euclidean']) /
pivot_nodes['euclidean']) * 100

    print("\nPercentage Difference in Runtime:\n", pivot_runtime)
    print("\nPercentage Difference in Nodes Expanded:\n", pivot_nodes)

    # Step 3: Summary Statistics by Map (Optional)
    map_summary = df.groupby(['map', 'heuristic']).agg({
        'avg_runtime_ms': ['mean', 'std'],
        'avg_nodes_expanded': ['mean', 'std']
    }).reset_index()
    map_summary.columns = ['map', 'heuristic', 'mean_runtime_ms', 'std_runtime_ms',
'mean_nodes_expanded', 'std_nodes_expanded']
    print("\nSummary by Map:\n", map_summary)

    # Step 4: Outlier Impact Statistics
    # With outliers
    grouped_with_outliers = df.groupby(['dataset_type', 'density_level', 'heuristic']).agg({
        'avg_runtime_ms': 'mean',
        'avg_nodes_expanded': 'mean'
    }).reset_index()

    # Without outliers
    no_outliers = df[(df['runtime_outlier'] == False) & (df['nodes_outlier'] == False)]
    grouped_no_outliers = no_outliers.groupby(['dataset_type', 'density_level',
'heuristic']).agg({
        'avg_runtime_ms': 'mean',
        'avg_nodes_expanded': 'mean'
    }).reset_index()

    # Merge and calculate percentage change
    outlier_impact = grouped_with_outliers.merge(grouped_no_outliers,
on=['dataset_type', 'density_level', 'heuristic'], suffixes=('_with', '_without'))
    outlier_impact['runtime_percent_change'] = ((outlier_impact['avg_runtime_ms_with'] -
outlier_impact['avg_runtime_ms_without']) / outlier_impact['avg_runtime_ms_with']) *
100
    outlier_impact['nodes_percent_change'] =
((outlier_impact['avg_nodes_expanded_with'] -
outlier_impact['avg_nodes_expanded_without']) /
outlier_impact['avg_nodes_expanded_with']) * 100
    print("\nOutlier Impact:\n", outlier_impact)

    # Step 5: Correlation Statistics
    # Remove any NaN values for correlation
    corr_data = df[['density', 'avg_runtime_ms', 'avg_nodes_expanded']].dropna()

    # Pearson's correlation
    pearson_density_runtime, _ = pearsonr(corr_data['density'],
corr_data['avg_runtime_ms'])
    pearson_density_nodes, _ = pearsonr(corr_data['density'],
corr_data['avg_nodes_expanded'])
    pearson_runtime_nodes, _ = pearsonr(corr_data['avg_runtime_ms'],
corr_data['avg_nodes_expanded'])

    print("\nCorrelation Statistics:")
    print(f"Pearson: Density vs Runtime: {pearson_density_runtime:.3f}")
    print(f"Pearson: Density vs Nodes Expanded: {pearson_density_nodes:.3f}")
    print(f"Pearson: Runtime vs Nodes Expanded: {pearson_runtime_nodes:.3f}")

    print("\nFinalizing output...")

    # Save summary for use in EE
    grouped_summary.to_csv('summary_stats.csv', index=False)
    print("\nSummary statistics saved as 'summary_stats.csv'")
```

Code Output

All collected and processed data was output into a CSV file with 1802 lines and hence a github repository is attached for reference:

https://github.com/disCodeOri/cs-ee-code-outputs

Raw Data (Moving AI Dataset)

Source:https://movingai.com/benchmarks/grids.html

Datasets:

1. Mazes with variable cooridor width - Nathan Sturtevant / HOG2 - https://movingai.com/benchmarks/maze/index.html
    - Maps:
        - maze512-8-0.map
        - maze512-8-1.map
        - maze512-8-2.map
        - maze512-16-0.map
        - maze512-16-1.map
        - maze512-16-2.map
        - maze512-32-0.map
        - maze512-32-1.map
        - maze512-32-2.map
    - Scenarios:
        - maze512-8-0.map.scen
        - maze512-8-1.map.scen

        - maze512-8-2.map.scen
        - maze512-16-0.map.scen
        - maze512-16-1.map.scen
        - maze512-16-2.map.scen
        - maze512-32-0.map.scen
        - maze512-32-1.map.scen
        - maze512-32-2.map.scen

2. Room maps - square rooms with random openings between them - Nathan Sturtevant / HOG2 - https://movingai.com/benchmarks/room/index.html
    - Maps:
        - 8room_000.map
        - 8room_001.map
        - 8room_002.map
        - 16room_000.map
        - 16room_001.map
        - 16room_002.map
        - 32room_000.map
        - 32room_001.map
        - 32room_002.map
    - Scenarios:
        - 8room_000.map.scen
        - 8room_001.map.scen
        - 8room_002.map.scen

- 16room_000.map.scen
- 16room_001.map.scen
- 16room_002.map.scen
- 32room_000.map.scen
- 32room_001.map.scen
- 32room_002.map.scen