Towards Better Heuristics: Optimizing A* Search with Insights from Beam Search & Uniform Cost Search

Daniel Gavrie Y. Clemente
De La Salle University Manila
2401 Taft Ave., Malate, Manila
1004 Metro Manila
+63 995 147 0254
daniel_clemente@dlsu.edu.ph

Dana Louise A. Guillarte
De La Salle University Manila
2401 Taft Ave., Malate, Manila
1004 Metro Manila
+63 949 992 8487
dana_guillarte@dlsu.edu.ph

Chrysille Grace L. So
De La Salle University Manila
2401 Taft Ave., Malate, Manila
1004 Metro Manila
+63 917 816 4723
chrysille_so@dlsu.edu.ph

ABSTRACT

This paper will focus on the features of the A Star Search as an advanced greedy search algorithm. The A Star(*) Search will be comparatively analyzed beside other related algorithms, such as the Uniform Cost Search (UCS) and Beam Search in terms of code implementation, time complexity, memory complexity, and optimality. The findings indicate that A* Search is generally the most time-efficient algorithm, followed by Beam Search, and UCS respectively. Beam search demonstrates lower and more stable memory efficiency, followed by A* Search and UCS. Overall, A* Search and Beam Search tend to have higher optimality due to their reliance on heuristic functions. This efficiency is directly proportional to the accuracy of heuristic values.

CCS Concepts

• **Advanced Algorithms** → **Greedy Search** →Heuristic Values

1. INTRODUCTION

People rely on search and pathfinding algorithms daily, whether navigating through a city, finding the quickest route in a map application, or even seeking efficient solutions in complex networks. These algorithms must be swift, precise, and efficient to provide accurate and timely information. The A* (A-star) Search algorithm is widely used in this context. A* Search is a greedy search algorithm designed to find the shortest path from a starting point to a destination by evaluating paths based on their cost and estimated future cost.

The A* algorithm combines the strengths of Dijkstra's algorithm and Greedy Best-First Search. It uses a heuristic function to estimate the cost from the current point to the goal while keeping track of the cost to reach the current point. This dual approach helps A* Search to efficiently explore the most promising paths while avoiding less optimal routes. Despite being efficient in pathfinding, the algorithm can still be improved in areas such as memory overhead and finding optimal solutions (Huang et al., 2022), where it underperforms compared to other heuristic algorithms.

This paper aims to explore and analyze the A* Search algorithm in detail, comparing it with two other advanced greedy search algorithms. By examining these alternatives, the paper will highlight the strengths and weaknesses of A* Search in various scenarios. Furthermore, it will delve into potential optimization strategies for enhancing the performance of the A* algorithm. Optimization possibilities include improvements in heuristic functions, better handling of dynamic and large-scale environments, and integration with other computational techniques to increase efficiency and effectiveness.

Through this comparison and exploration of optimization techniques, the paper seeks to provide a comprehensive understanding of A* Search and its place within the broader landscape of pathfinding algorithms. This will offer valuable insights into how to leverage A* Search in practical applications and how to refine its performance for specific use cases.

2. FORMAL DEFINITION OF THE A* ALGORITHM

The A* search algorithm is an advanced informed search algorithm widely used for finding the shortest path between nodes in a graph. It combines Dijkstra's algorithm and best-first search by using the cost to reach a node $(g(n))$ and a heuristic estimate of the cost to reach the goal $(h(n))$. The algorithm evaluates nodes with the function $f(n) = g(n) + h(n)$ selecting nodes with the lowest $f(n)$ represents value for expansion, thus prioritizing promising paths.

The efficiency of A* hinges on the quality of the heuristic function, which must be admissible (not overestimating the true cost) and consistent (ensuring the heuristic value decreases along a path). Recent advancements have focused on improving A*'s efficiency and robustness in complex environments. For example, Lou et al. (2021) introduced the EBS-A* algorithm, which enhances path planning by incorporating expansion distance, bidirectional search, heuristic function optimization, and path smoothing. Zhang and Zhao (2024) developed an improved A* algorithm for inspection robots, achieving significant reductions in planning time and path length.

These advancements demonstrate the ongoing evolution of the A* algorithm and highlight the need for continuous improvements, reinforcing its role in real-time path planning across various applications, from robotics to logistics.

3. ANALYSIS AND COMPARISON OF THE A* ALGORITHM TO RELATED ALGORITHMS

3.1 Methodology

3.1.1 Graph Setup

In order to effectively compare A* Search, Uniform Cost Search, and Beam Search with each other, each advanced algorithm was implemented as a Python class. These algorithms are called one by one through the main file that includes the graph node, edge, and heuristic values.

The sample graph is based on the structure of Metro Manila, where the nodes are labeled as the different cities. For demonstration purposes, the goal node is fixed to Parañaque, to ensure that the heuristic values are used properly.

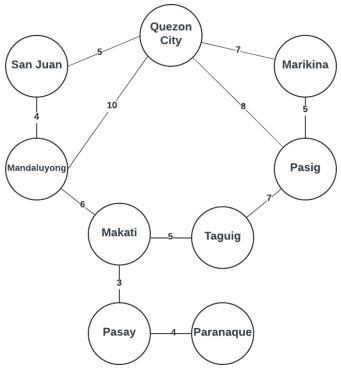

Figure 1. Sample Graph of Metro Manila

```
#Define a new graph with cities in Metro Manila
metro_manila_graph = {
    "Makati": {"Taguig": 5, "Pasay": 3, "Mandaluyong": 6},
    "Taguig": {"Makati": 5, "Pasig": 7},
    "Pasay": {"Makati": 3, "Paranaque": 4},
    "Mandaluyong": {"Makati": 6, "San Juan": 4, "Quezon
City": 10},
    "Pasig": {"Taguig": 7, "Marikina": 5, "Quezon City":
8},
    "San Juan": {"Mandaluyong": 4, "Quezon City": 5},
    "Quezon City": {"Mandaluyong": 10, "San Juan": 5,
"Pasig": 8, "Marikina": 7},
    "Marikina": {"Pasig": 5, "Quezon City": 7},
    "Paranaque": {"Pasay": 4}
}

# Updated heuristic values assuming "Paranaque" as the goal
heuristic_values = {
    "Makati": 6,
    "Taguig": 9,
    "Pasay": 4,
    "Mandaluyong": 10,
    "Pasig": 15,
    "San Juan": 12,
    "Quezon City": 17,
    "Marikina": 20,
    "Paranaque": 0    # Paranaque is the goal
}
```

Figure 2. Declaration of the Sample Graph and Heuristic Values in Python

Time and malloc libraries were used to measure the execution time and memory usage for each algorithm run. The following corresponding values are recorded and displayed: final path, traversed path, total cost, (execution) time, nodes expanded, max frontier, memory usage. Final path contains the straightforward path generated from the search, while the

traversed path will contain all the nodes explored regardless of its inclusion in the final path.

```
def run_search(search_algorithm, start_city=None,
end_city=None):
    tracemalloc.start()
    start_time = time.perf_counter()
    visited_order, total_cost, path, nodes_expanded,
max_frontier_size, visit_count =
search_algorithm.search(start_city, end_city)

    end_time = time.perf_counter()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    elapsed_time = (end_time - start_time) * 1e6  # Convert
to microseconds
    path_str = " -> ".join(path)
    visited_str = " -> ".join(visited_order)

    result_text = (
        f"Final Path: {path_str}\n"
        f"Path Traversed: {visited_str}\n"
        f"Total Cost: {total_cost}\n"
        f"Time: {elapsed_time:.4f} microseconds\n"
        f"Nodes Expanded: {nodes_expanded}\n"
        f"Max Frontier Size: {max_frontier_size}\n"
        f"Memory Usage: Current={current / 1024:.2f}KB,
Peak={peak / 1024:.2f}KB\n"
        f"Visit Count: {visit_count}"
    )

    print(result_text)
```

Figure 3. Implementation of Running the Search Algorithms in Python

The side-by-side comparison is done by calling each algorithm consecutively in a main file.

*3.1.2 A**

The code for A* Search algorithm is as shown below.

```
import heapq

class AStarSearch:
    def __init__(self, graph, heuristic_values):
        self.graph = graph
        self.heuristic_values = heuristic_values

    def search(self, start, goal):
        frontier = []
        heapq.heappush(frontier, (0, start))

        visit_count = {start: 1}
        max_frontier_size = 0
        cost_so_far = {start: 0}
        function_values = {start:
self.heuristic_values.get(start, 0)}
        came_from = {start: None}
        visited_order = []

        while frontier:
            max_frontier_size = max(max_frontier_size,
len(frontier))
            current_cost, current_node =
heapq.heappop(frontier)
            visited_order.append(current_node)

            if current_node == goal:
                path = []
                while current_node is not None:
                    path.append(current_node)
                    current_node = came_from[current_node]
                return visited_order, cost_so_far[goal],
path[::-1], len(visited_order), max_frontier_size,
visit_count

            for neighbor, cost in
self.graph[current_node].items():
                new_cost = cost_so_far[current_node] + cost
                neighbor_cost = cost_so_far.get(neighbor,
float('inf'))

                if new_cost < neighbor_cost:
                    cost_so_far[neighbor] = new_cost
                    function_values[neighbor] = new_cost +
self.heuristic_values.get(neighbor, 0)
                    heapq.heappush(frontier,
(function_values[neighbor], neighbor))
                    came_from[neighbor] = current_node
                    visit_count[neighbor] =
visit_count.get(neighbor, 0) + 1

        return visited_order, float('inf'), [],
len(visited_order), max_frontier_size, visit_count
```

Figure 4. Implementation of A* in Python

The code initializes a priority queue, frontier, with the start node at priority zero, using Python's heapq for efficient retrieval of nodes with the lowest function cost. It maintains dictionaries for actual costs (cost_so_far), A* function values (function_values), parent nodes (came_from), and node visit counts (visit_count). It also tracks the maximum size of the frontier (max_frontier_size) and the order in which nodes are visited (visited_order).

During the search, the algorithm extracts the node with the lowest function value from the frontier. If this node is the goal, it reconstructs and returns the path and relevant statistics. For each neighbor, it calculates a new path cost and updates records if the new path is better. If no path is found, it returns an empty path and infinity for the cost.

Nodes are managed in "unvisited" and "visited" lists, with the start node's g-score initialized to zero and its f-score based on a heuristic function. The algorithm picks the node with the lowest f-score from the "unvisited" list, examines neighbors, updates costs and records for better paths, and moves fully explored nodes

to the "visited" list. The visited_order list records the order of visited nodes, and max_frontier_size tracks the largest frontier size. If the goal is found, the path is traced back using "previous node" pointers. If the "unvisited" list is empty and no path is found, it indicates no path exists between the start and goal.

The function returns visited_order, cost_so_far[goal], the reconstructed path, the length of visited_order, max_frontier_size, and visit_count.

3.1.3 Uniform Cost Search

The Uniform Cost Search (UCS) algorithm finds the shortest path in a graph with non-negative edge weights by exploring nodes in order of increasing path cost from a starting node, ensuring the first goal node reached has the shortest possible path.

In this paper, the implementation of UCS is shown below:

```
import heapq

class UniformCostSearch:
    def __init__(self, graph):
        self.graph = graph

    def search(self, start, goal):
        frontier = []
        heapq.heappush(frontier, (0, start))

        visit_count = {start: 1}
        max_frontier_size = 0
        cost_so_far = {start: 0}
        came_from = {start: None}
        visited_order = []

        while frontier:
            max_frontier_size = max(max_frontier_size,
len(frontier))
            current_cost, current_node =
heapq.heappop(frontier)
            visited_order.append(current_node)

            if current_node == goal:
                path = []
                while current_node is not None:
                    path.append(current_node)
                    current_node = came_from[current_node]
                return visited_order, cost_so_far[goal],
path[::-1], len(visited_order), max_frontier_size,
visit_count

            for neighbor, cost in
self.graph[current_node].items():
                new_cost = current_cost + cost
                neighbor_cost = cost_so_far.get(neighbor,
float('inf'))

                if new_cost < neighbor_cost:
                    cost_so_far[neighbor] = new_cost
                    heapq.heappush(frontier, (new_cost,
neighbor))
                    came_from[neighbor] = current_node
                    visit_count[neighbor] =
visit_count.get(neighbor, 0) + 1

        return visited_order, float('inf'), [],
len(visited_order), max_frontier_size, visit_count
```

Figure 5. Implementation of UCS in Python

The UCS algorithm begins by initializing a priority queue, frontier, with the start node set to a priority of zero, using Python's heapq for efficient retrieval of the lowest-cost nodes. It maintains dictionaries for tracking the actual costs (cost_so_far), parent nodes (came_from), and visit counts (visit_count). Additionally, it tracks the maximum size of the frontier (max_frontier_size) and the order of visited nodes (visited_order).

During the search, the algorithm extracts the node with the lowest cost from the frontier. If this node is the goal, it reconstructs the path from start to goal, returning the path, total cost, visited order, number of visited nodes, maximum frontier size, and visit counts. For each neighbor, it calculates a new path cost and updates the records if the new path is more efficient, incrementing the visit count. If the goal is unreachable, the algorithm returns an empty path and an infinite cost.

UCS explores nodes based solely on known travel costs, processing nodes in order of lowest cumulative cost. The visited_order list provides a record of the visited sequence, while max_frontier_size reflects the peak size of the frontier. If the goal is reached, the path is reconstructed using came_from; otherwise, the absence of a viable path is indicated. The algorithm ultimately returns the visited order, total cost, path, number of nodes visited, maximum frontier size, and visit counts.

3.1.4 Beam Search

Beam Search is a variant of a breadth-first search algorithm. The algorithm searches a weighted graph to find the optimal path (*AI | Search Algorithms | BEAM Search | CodeCademy*, 2023). The algorithm is mainly used in Natural Language Processing and speech

recognition (*What Is Beam Search? Explaining the Beam Search Algorithm | Width.ai*, n.d.).

```
class BeamSearch:
    def __init__(self, graph, heuristic_values,
beam_width):
        self.graph = graph
        self.heuristic_values = heuristic_values
        self.beam_width = beam_width

    def search(self, start_node, end_node):
        # Initialize the frontier with the start city
        frontier = [(start_node, 0)]
        visited_order = []
        nodes_expanded = 0
        max_frontier_size = 1
        visit_count = {node: 0 for node in self.graph}

        while frontier:
            # Sort the frontier based on the heuristic
values and limit its size to the beam width
            frontier.sort(key=lambda x:
self.heuristic_values[x[0]], reverse=True)
            frontier = frontier[:self.beam_width]

            # Pop the city with the highest heuristic value
from the frontier
            current_node, current_cost = frontier.pop(0)
            visited_order.append(current_node)
            visit_count[current_node] += 1

            # If the current city is the end city, return
the results
            if current_node == end_node:
                return visited_order, current_cost,
visited_order, nodes_expanded, max_frontier_size,
visit_count

            # Expand the current city
            nodes_expanded += 1
            for neighbor, cost in
self.graph[current_node].items():
                if neighbor not in visited_order:
                    frontier.append((neighbor, current_cost
+ cost))

            # Update the maximum frontier size
            max_frontier_size = max(max_frontier_size,
len(frontier))

        # If no path is found, return None
        return None
```

Figure 6. Implementation of Beam Search in Python

The algorithm initializes the frontier with the starting node and sets up variables to track the visited order, nodes expanded, maximum frontier size, and visit count for each node in the graph. Next, the algorithm enters a while loop to iterate through the frontier and expand the most promising nodes based on the heuristic values. It sorts the frontier based on heuristic values and limits its size to the beam width. Then, it pops the node with the highest heuristic value from the frontier, updates the visited order and visit count, and checks if the current node is the end node. If the current node is not the end node, it expands the current node by adding its neighbors to the frontier and updates the maximum frontier size. The algorithm continues this process until it finds the end node or exhausts all possible paths.

3.2 Program Outputs

All three advanced algorithms were tested on varying test cases consecutively. The command line outputs are as listed below.

Test Case 1: Parañaque to San Juan

```
Enter the start city: Paranaque
Enter the end city: San Juan

Uniform Cost Search:
Final Path: Paranaque -> Pasay -> Makati -> Mandaluyong ->
San Juan
Path Traversed: Paranaque -> Pasay -> Makati -> Taguig ->
Mandaluyong -> San Juan
Total Cost: 17
Time: 1716.2000 microseconds
Nodes Expanded: 6
Max Frontier Size: 3
Memory Usage: Current=0.39KB, Peak=0.89KB
Visit Count: {'Paranaque': 1, 'Pasay': 1, 'Makati': 1,
'Taguig': 1, 'Mandaluyong': 1, 'Pasig': 1, 'San Juan': 1,
'Quezon City': 1}

A* Search:
Final Path: Paranaque -> Pasay -> Makati -> Mandaluyong ->
San Juan
Path Traversed: Paranaque -> Pasay -> Makati -> Taguig ->
Mandaluyong -> San Juan
Total Cost: 17
Time: 176.0000 microseconds
Nodes Expanded: 6
Max Frontier Size: 3
Memory Usage: Current=0.30KB, Peak=1.02KB
Visit Count: {'Paranaque': 1, 'Pasay': 1, 'Makati': 1,
'Taguig': 1, 'Mandaluyong': 1, 'Pasig': 1, 'San Juan': 1,
'Quezon City': 1}

Beam Search:
Final Path:
Path Traversed: Paranaque -> Pasay -> Makati -> Mandaluyong
-> Quezon City -> Marikina -> Pasig -> Marikina -> Pasig ->
Taguig
Total Cost: inf
Time: 324.9000 microseconds
Nodes Expanded: 10
Max Frontier Size: 4
Memory Usage: Current=0.33KB, Peak=0.74KB
Visit Count: {'Makati': 1, 'Taguig': 1, 'Pasay': 1,
'Mandaluyong': 1, 'Pasig': 2, 'San Juan': 0, 'Quezon City':
1, 'Marikina': 2, 'Paranaque': 1}
```

Figure 7. Test Case 1 Outputs in Python

Test Case 2: Parañaque to Marikina

```
Enter the start city: Paranaque
```

```
Enter the end city: Marikina

Uniform Cost Search:
Final Path: Paranaque -> Pasay -> Makati -> Taguig -> Pasig
-> Marikina
Path Traversed: Paranaque -> Pasay -> Makati -> Taguig ->
Mandaluyong -> San Juan -> Pasig -> Quezon City -> Quezon
City -> Marikina
Total Cost: 24
Time: 940.6000 microseconds
Nodes Expanded: 10
Max Frontier Size: 3
Memory Usage: Current=0.65KB, Peak=1.12KB
Visit Count: {'Paranaque': 1, 'Pasay': 1, 'Makati': 1,
'Taguig': 1, 'Mandaluyong': 1, 'Pasig': 1, 'San Juan': 1,
'Quezon City': 2, 'Marikina': 1}

A* Search:
Final Path: Paranaque -> Pasay -> Makati -> Taguig -> Pasig
-> Marikina
Path Traversed: Paranaque -> Pasay -> Makati -> Taguig ->
Mandaluyong -> San Juan -> Pasig -> Quezon City -> Quezon
City -> Marikina
Total Cost: 24
Time: 134.5000 microseconds
Nodes Expanded: 10
Max Frontier Size: 3
Memory Usage: Current=0.50KB, Peak=1.20KB
Visit Count: {'Paranaque': 1, 'Pasay': 1, 'Makati': 1,
'Taguig': 1, 'Mandaluyong': 1, 'Pasig': 1, 'San Juan': 1,
'Quezon City': 2, 'Marikina': 1}

Beam Search:
Final Path: Paranaque -> Pasay -> Makati -> Mandaluyong ->
Quezon City -> Marikina
Path Traversed: Paranaque -> Pasay -> Makati -> Mandaluyong
-> Quezon City -> Marikina
Total Cost: 30
Time: 120.6000 microseconds
Nodes Expanded: 5
Max Frontier Size: 4
Memory Usage: Current=0.31KB, Peak=0.74KB
Visit Count: {'Makati': 1, 'Taguig': 0, 'Pasay': 1,
'Mandaluyong': 1, 'Pasig': 0, 'San Juan': 0, 'Quezon City':
1, 'Marikina': 1, 'Paranaque': 1}
```

Figure 8. Test Case 2 Outputs in Python

Test Case 3: Parañaque to Quezon City

```
Enter the start city: Paranaque
Enter the end city: Quezon City

Uniform Cost Search:
Final Path: Paranaque -> Pasay -> Makati -> Mandaluyong ->
San Juan -> Quezon City
Path Traversed: Paranaque -> Pasay -> Makati -> Taguig ->
Mandaluyong -> San Juan -> Pasig -> Quezon City
Total Cost: 22
Time: 1798.9000 microseconds
Nodes Expanded: 8
Max Frontier Size: 3
Memory Usage: Current=0.59KB, Peak=1.09KB
Visit Count: {'Paranaque': 1, 'Pasay': 1, 'Makati': 1,
'Taguig': 1, 'Mandaluyong': 1, 'Pasig': 1, 'San Juan': 1,
'Quezon City': 2, 'Marikina': 1}

A* Search:
Final Path: Paranaque -> Pasay -> Makati -> Mandaluyong ->
San Juan -> Quezon City
Path Traversed: Paranaque -> Pasay -> Makati -> Taguig ->
Mandaluyong -> San Juan -> Pasig -> Quezon City
Total Cost: 22
Time: 159.7000 microseconds
Nodes Expanded: 8
Max Frontier Size: 3
Memory Usage: Current=0.44KB, Peak=1.14KB
Visit Count: {'Paranaque': 1, 'Pasay': 1, 'Makati': 1,
'Taguig': 1, 'Mandaluyong': 1, 'Pasig': 1, 'San Juan': 1,
'Quezon City': 2, 'Marikina': 1}

Beam Search:
Final Path: Paranaque -> Pasay -> Makati -> Mandaluyong ->
Quezon City
Path Traversed: Paranaque -> Pasay -> Makati -> Mandaluyong
-> Quezon City
Total Cost: 23
Time: 149.6000 microseconds
Nodes Expanded: 4
Max Frontier Size: 3
Memory Usage: Current=0.30KB, Peak=0.63KB
Visit Count: {'Makati': 1, 'Taguig': 0, 'Pasay': 1,
'Mandaluyong': 1, 'Pasig': 0, 'San Juan': 0, 'Quezon City':
1, 'Marikina': 0, 'Paranaque': 1}
```

Figure 9. Test Case 3 Outputs in Python

3.3 Time Complexity

The execution times of Uniform Cost Search (UCS), A* Search, and Beam Search exhibit notable differences, as measured in microseconds to highlight these variations. Utilizing microseconds enables a more accurate comparison, given the graph's compact dimensions, which would yield negligible fluctuations if measured in seconds.

Test	UCS	A* Search	Beam Search
1	1716.20 µs	176.00 µs	324.90 µs
2	940.60 µs	134.50 µs	120.60 µs
3	1798.90 µs	159.70 µs	149.60 µs

Figure 10. Execution Times of Each Search Algorithm Per Test Case

The findings indicate that A* Search is the most time-efficient algorithm, followed by Beam Search and UCS. A* Search's effectiveness stems from its heuristic approach, which guides the search more efficiently than UCS, which lacks heuristic use, and Beam Search, which may explore more nodes depending on the beam width.

Detailed analysis of expanded nodes and frontier size reveals performance variations. In test cases where the goal node is closer to the

start node, both UCS and A* consistently expanded fewer nodes and maintained a smaller frontier size than Beam Search. However, in cases where the goal node is significantly farther from the start node, Beam Search expanded fewer nodes than the other two algorithms but had a similar frontier size. For instance, A* Search and UCS expanded 6 nodes in Test Case 1, 10 and 9 nodes in Test Case 2, and 8 nodes in Test Case 3, with a maximum frontier size of 3. In contrast, Beam Search expanded 10, 5, and 4 nodes, with a larger maximum frontier size, indicating its higher time consumption and lower efficiency in some scenarios.

Regarding time complexity, both UCS and A* Search generally exhibit $O(b^d)$ in the worst case, with b as the branching factor and d as the depth of the solution. A* Search often outperforms UCS due to its heuristic, which reduces node expansion. Beam Search, with a complexity of $O(b^{wd})$, where w is the beam width, can be faster with an optimal beam width. However, an excessively large beam width can lead to increased node exploration and higher time consumption, as seen in the test cases.

3.4 Memory Complexity

Test	UCS		A* Search		Beam Search	
	Current	Peak	Current	Peak	Current	Peak
1	0.39KB	0.89 KB	0.30KB	1.02 KB	0.33KB	0.74 KB
2	0.65KB	1.12 KB	0.50KB	1.20 KB	0.31KB	0.74 KB
3	0.59KB	1.09 KB	0.44KB	1.14 KB	0.30KB	0.63 KB

Figure 11. Memory Usage of Each Search Algorithm Per Test Case

Among the three algorithms, Beam Search consistently demonstrates lower and more stable memory usage compared to UCS and A* Search. In Test Case 1, Beam Search used 0.33KB of memory, peaking at 0.74KB, while UCS and A* Search had higher peaks of 0.89KB and 1.02KB, respectively. In Test Case 2, Beam Search reached a peak of 0.74KB, whereas UCS and A* Search peaked at 1.12KB and 1.20KB. In Test Case 3, Beam Search again had the lowest memory usage, with current and peak values of 0.30KB and 0.63KB. Beam Search's consistent memory efficiency is due to its fixed beam width, which limits the number of nodes stored in memory.

Additionally, Beam Search maintains relatively low memory usage compared to UCS and A* Search, despite variations in frontier size and the number of nodes expanded. For instance, in Test Case 1, Beam Search expanded 10 nodes, while UCS and A* Search each expanded 6 nodes. In Test Case 2, Beam Search expanded 5 nodes, compared to 10 for UCS and 9 for A* Search. In Test Case 3, Beam Search expanded 4 nodes, while UCS and A* Search each expanded 8 nodes. Beam Search's ability to effectively prune the search space helps keep memory usage manageable. Specifically, Beam Search's peak memory usage was 0.74KB in Test Cases 1 and 2, and 0.63KB in Test Case 3, while UCS and A* Search consistently exhibited higher peak memory usages.

In terms of memory complexity, UCS and A* Search generally require $O(b^d)$ memory, where b is the branching factor and d is the solution depth. Beam Search, with a complexity of $O(bwd)$, where w is the beam width, can have lower memory requirements due to the beam width limitation. This can be advantageous, especially when the beam width is well-calibrated, balancing search breadth with memory consumption. However, an excessively large beam width can increase memory usage, making the choice of beam width crucial for Beam Search's efficiency in memory-constrained environments.

3.5 Limitations and Recommendations

The study compares A* Search, Uniform Cost Search (UCS), and Beam Search, highlighting their unique strengths and weaknesses. A* Search is highly efficient when an accurate heuristic is available, balancing path cost and heuristic estimates, but can suffer from high memory usage in complex graphs. UCS

excels in scenarios where minimizing path cost is crucial, as it explores the least costly path without heuristics. Beam Search is noted for its memory efficiency, controlling the number of paths explored, making it ideal for memory-constrained, real-time applications.

While modifications like integrating Beam Search traits and weighted heuristics show potential, these require thorough testing and academic validation. Preliminary suggestions indicate that A* Search, despite being the most time-efficient, faces challenges such as high memory complexity, issues with handling multiple agents, and dependency on heuristic functions. Beam Search's lower and more stable memory usage suggests that incorporating its controlled exploration features could reduce A* Search's memory footprint while maintaining its advantages. These insights, as supported by Foead et al. (2021), provide a foundation for further research and development in optimizing search algorithms.

Several enhancements can optimize the A* Search algorithm. One approach involves refining the heuristic function by adjusting the heuristic weight to balance speed and accuracy:

```
def modified_heuristic(node, goal):
    return heuristic(node, goal) * weight_factor

weight_factor = 1.2  # A factor greater than 1 biases
towards quicker but potentially less optimal paths
```

Figure 12. Implementation of Modified Heuristic in Python

Another improvement is refining the cost function to focus on detailed cost assessments, minimizing the exploration of non-optimal routes (Yiu et al., 2018):

```
def cost_function(node, neighbor):
    return path_cost[node] + edge_cost[node][neighbor] +
heuristic(neighbor, goal)

def priority_function(node):
    return cost_function(node) * priority_weight

priority_weight = 0.8  # A factor less than 1 biases
towards exploring more promising paths
```

Figure 13. Implementation of Cost Function Refinement in Python

Introducing a beam width-like mechanism from Beam Search can help control memory usage by limiting the number of nodes expanded at each step:

```
def beam_search_expansion(frontier, beam_width):
    return sorted(frontier, key=lambda x: cost_function(x,
goal))[:beam_width]

beam_width = 5  # Only keep the top 5 nodes based on the
cost function
```

Figure 14. Implementation of Beam Width-Like Mechanism for A* in Python

Additionally, incorporating manual search markers to avoid known obstacles or traps can enhance search efficiency (Junfeng et al., 2009):

```
obstacle_markers = set([('NodeA', 'NodeB'), ('NodeC',
'NodeD')]) # values are just examples and should be
modified depending on the graph input

def is_obstacle(node_pair):
    return node_pair in obstacle_markers

def modified_expansion(current_node):
    for neighbor in neighbors(current_node):
        if not is_obstacle((current_node, neighbor)):
            expand_node(current_node, neighbor)
```

Figure 15. Implementation of Manual Search or Obstacle Markers in Python

These recommendations aim to address A* Search's current limitations and enhance its performance across various scenarios. The proposed adjustments focus on improving time and memory efficiency by integrating traits from Beam Search and refining heuristic functions. However, these suggestions are preliminary and require further testing and rigorous academic research to confirm their effectiveness in practical applications.

4. APPLICATIONS OF A* SEARCH

4.1 Video Games: DOTA

A* Search, renowned for its efficiency and effectiveness, finds significant applications in video games, particularly in multiplayer online battle arena (MOBA) games like DOTA (Defense of the Ancients). In DOTA, players navigate complex terrains using various characters, and A* Search helps to optimize these movements. The algorithm's ability to find the shortest path between points on a map, even when considering obstacles and varied terrains, makes it invaluable for ensuring smooth and strategic navigation. This foundational use of A* Search extends

beyond DOTA to other MOBA games, highlighting its critical role in enhancing gameplay by enabling characters to traverse maps efficiently, thereby contributing to the game's strategic depth and playability (Medium, 2023).

4.2 Network routing

A* Search is also integral to network routing, where its application ensures efficient data transmission across complex networks. When data packets travel from one computer to another, A* Search helps routers determine the most efficient path by incorporating network conditions and historical data on packet transfers. This heuristic-based approach allows routers to dynamically adapt to changing network conditions, ensuring that data reaches its destination in the least possible time. While the paths chosen may not always be the absolute shortest due to the dynamic nature of networks, the reliability and efficiency of A* Search in routing algorithms ensure robust and timely communication across the internet, demonstrating its versatility and importance in both digital entertainment and essential network infrastructure (Medium, 2023).

4.3 Robotics

A* Search, a robust algorithm, is now a staple in the arsenal of robots, from Roombas to self-driving cars. These robots leverage the pathfinding algorithm to deftly navigate obstacles and reach their goals (*A* Algorithm in AI (a* Search Algorithm)*, 2024). The use of A* Search in robot path planning is a testament to its efficiency, enabling these machines to move around in the shortest path possible, saving time and resources (Guruji et al., 2016, p. 3).

4.4 GPS Navigation

A* Search plays a crucial role in navigation, particularly in GPS systems. The algorithm's primary function is to find the shortest path for the user from one location to another. It does this by considering real-time traffic data and road conditions, ensuring the user is guided along the most efficient route (*A* Algorithm in AI (a* Search Algorithm)*, 2024).

5. CONCLUSION

The study aimed to investigate and enhance the A* Search algorithm, commonly used in pathfinding. It compared A* Search with Uniform Cost Search (UCS) and Beam Search, highlighting their strengths and weaknesses. A* Search is effective with accurate heuristics but can be memory-intensive and less efficient in complex graphs or multi-agent scenarios. UCS excels at minimizing path costs without heuristics, making it suitable when cost is the primary concern. Beam Search is noted for its memory efficiency, ideal for real-time applications with limited resources.

Incorporating traits from Beam Search, like controlled node expansion, could reduce A* Search's memory usage while retaining heuristic benefits. Additionally, refining heuristic and cost functions can improve performance by balancing speed and accuracy. However, these recommendations require extensive testing and validation.

The study underscores the importance of selecting the right algorithm based on specific contexts and the potential to enhance A* Search through targeted refinements.

6. REFERENCES

[1] A* algorithm in AI (A* search algorithm). (2024, July 20). AlmaBetter. https://www.almabetter.com/bytes/tutorials/artificial-intelligence/a-star-algorithm-in-ai

[2] A* Search Real-Life Application. *Medium*, Oct. 10, 2023. https://medium.com/@aglubagerry/a-search-real-life-application-2bd175624952

[3] *AI | Search Algorithms | BEAM Search | CodeCademy*. (2023, June 5). Codecademy. https://www.codecademy.com/resources/docs/ai/search-algorithms/beam-search

[4] What is Beam Search? Explaining The Beam Search Algorithm | Width.ai. (n.d.). https://www.width.ai/post/what-is-beam-search

[5] Foead, D., Ghifari, A., Kusuma, M. B., Hanafiah, N., & Gunawan, E. (2021). A Systematic Literature review of A*

pathfinding. Procedia Computer Science, 179, 507–514. https://doi.org/10.1016/j.procs.2021.01.034
[6] Guruji, A. K., Agarwal, H., & Parsediya, D. (2016). Time-efficient A* algorithm for robot path planning. Procedia Technology, 23, 144–149. https://doi.org/10.1016/j.protcy.2016.03.010
[7] Junfeng, Y., Binbin, Z., & Qingda, Z. The Optimization of A* Algorithm in the Practical Path Finding Application. In 2009 WRI World Congress on Software Engineering.
[8] Lou, S., Jing, J., He, H., & Liu, W. (2021). An Efficient and Robust Improved A* Algorithm for Path Planning. Symmetry, 13(11), 2213. https://doi.org/10.3390/sym13112213
[9] Yiu, Y. F., Du, J., & Mahapatra, R. (2018, September). Evolutionary heuristic a* search: Heuristic function optimization via genetic algorithm. In 2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE) (pp. 25-32). IEEE.
[10] Zhang, Y., & Zhao, Q. (2024). Complex Environment Based on Improved A* Algorithm Research on Path Planning of Inspection Robots. Processes, 12(5), 855. https://doi.org/10.3390/pr12050855

7. APPENDIX A: RECORD OF CONTRIBUTION

Activity	Daniel Gavriel Y. Clemente	Dana Louise A. Guillarte	Chrysille Grace L. So
Topic Formulation	27	40	33
Definition of the Problem	30	40	30
Methodology	25	25	50
Coding of Algorithms	33	32	33
Analysis Results and Discussion	31	53	16
Applications	55	10	35
Raw Total	201	201	201
TOTAL	**33.33**	**33.33**	**33.33**

Daniel Gavrie Y. Clemente
July 29, 2024

Dana Louise A. Guillarte
July 29, 2024

Chrysille Grace L. So
July 29, 2024