Assignment 2a
Tree-Based Search

Authors

Truong Ngoc Gia Hieu	105565520
Duong Nguyen Dang	105508444
Huynh Trong Hieu	105551833
Pham Nguyen Minh Hoang	105543500

TABLE OF CONTENT

Part A: Abstract

1.1 Assignment Requirements

In today's globalized world, the rapid advancement of computational intelligence has transformed the way humans approach to solving complex problems. Therefore, Swinburne University of Technology introduces the Introduction to Artificial Intelligence (COS30019) course, providing students with an in-depth understanding of algorithmic problem-solving techniques and the sophisticated concepts of artificial intelligence employed to tackle intricate problems. Moreover, the unit covers the ethical issues and security risks associated with AI and introduces students to a range of principles and frameworks targeting these issues.

Particularly, the second assignment aims to challenge students to translate sophisticated theories in classrooms into practical and functional software solutions. By using real problems, including the Route Finding Problem, the assignment focuses on the hands-on application of both uninformed and informed tree-based search strategies within a 2D spatial environment. After completing the assignment, the team is expected to deliver a robust Python-based application that demonstrates algorithmic precision, specifically in adhering to strict tie-breaking and node expansion rules. Finally, the fascinating teamwork involved in this project serves as a vital platform for members to synchronize their technical skills, guaranteeing the development of an integrated system where the code is not only efficient but also remarkably easy to understand.

1.2 Team Introduction

The Gamble team comprises four dedicated students with a shared commitment to technical excellence and collaborative success in the unit. For the first part of the second assignment, team members' efforts were directed toward engineering a sophisticated tree-based search framework to navigate the Route Finding Problem. By ensuring the workflow remains on the right track, the team leader decided to divide the project into smaller steps, including algorithmic analysis and selection, data structure architecture, system implementation, and rigorous quality assurance. Following these steps, the team leader ensures that all steps are executed with precision and seamlessly integrated into a unified system.

Name	Student ID	Role and Responsibilities
Truong Ngoc Gia Hieu	105565520	Leader: Project coordination, algorithmic analysis, and final report consolidation.
Huynh Trong Hieu	1055518333	Vice-leader and system developer: Assisting in project management, designing data structures, and implementing search algorithms.

Duong Nguyen Dang	105508444	AI Specialist: Developing informed search strategies (A*, GBFS) and custom heuristic methods (CUS1, CUS2).
Pham Nguyen Minh Hoang	105543500	Quality Assurance: System testing, cross-validating tie-breaking logic, and documentation review.

1.3 Problem Statement

The primary purpose of the first part of the second assignment is to address the Route Finding Problem by transforming theory lessons into practical coding and logical thinking. Particularly, the problem involves students programming an agent from a starting location to a target goal within a 2D environment where the landscape is represented as a network of nodes and weighted edges. Based on the assignment's requirements, the team must solve the following core challenges in order to achieve an optimal solution:

- Algorithmic Breadth: By using provided materials, including textbooks and slides, students should implement six different search strategies, ranging from Uninformed Search (DFS, BFS, and CUS1) to Informed Search (GBFS, A*, and CUS2).
- Expansion and Tie-breaking Precision: The assignment asks students to ensure the search results are predictable and consistent, which means that the agent must follow strict rules. Furthermore, the team must program the agent to always prioritize nodes based on their unique IDs and the order they were discovered when facing multiple paths of equal cost, ensuring our search framework remains deterministic and eliminating randomness, which is crucial for verifying the correctness of each search path.
- Heuristic Accuracy: Designing and applying effective heuristic functions, such as Manhattan and Euclidean distance, for informed search methods to minimize total path costs and improve efficiency.
- System Integrity: Guaranteeing the final implementation is robust across various map configurations while maintaining a codebase that is easy to understand and well-documented.

In the next part, the report dives into the core content: "Tree-Based Search". Additionally, the section will demonstrate how our team transforms theories into practical applications, generating a system that is not only robust and deterministic but also ensures the final code is easy to understand for any developer or reviewer.

Part B: Tree-Based Search

In the second section, the report provides a comprehensive overview of the Tree-Based Search methodology for navigating a 2D environment. This framework is selected for its systematic approach to state space exploration, maintaining a hierarchical structure that remains efficient and easy to understand. In addition, this part details the transition from basic Uninformed Search strategies to advanced Informed Search algorithms, including A* and GBFS. By following a consistent framework, the Gamble team ensures that each agent's behavior remains robust and deterministic, ultimately identifying the most optimal route.

2.1. Methodology Overview

2.1.1 Framework Introduction

According to DeepMind, the integration of tree based search framework with deep learning in AlphaGo allowed the system to explore vast decision spaces and defeat world champion Lee Sedol, which indicates the strategic power of search frameworks in complex environments. Furthermore, this profound solution is significantly crucial in modern areas, including robotics for path planning and autonomous vehicle navigation, where finding the most efficient route within a 2D environment is critical. As a result, by utilizing suitable approaches, our team ensures the agent can handle spatial complexity with logic that remains robust, deterministic, and easy to understand by utilizing innovative solutions. In addition, the framework also supports dynamic replanning, enabling it to adapt to varying graph topologies and offering clear tradeoffs between computational cost and solution quality through algorithmic selection. As my team expected, the agent can both reliably compute viable routes and balance exploration depth, memory usage, and optimality at the same time based on the requirements because navigation is structured as a systematic tree expansion.

2.1.2 Core Components

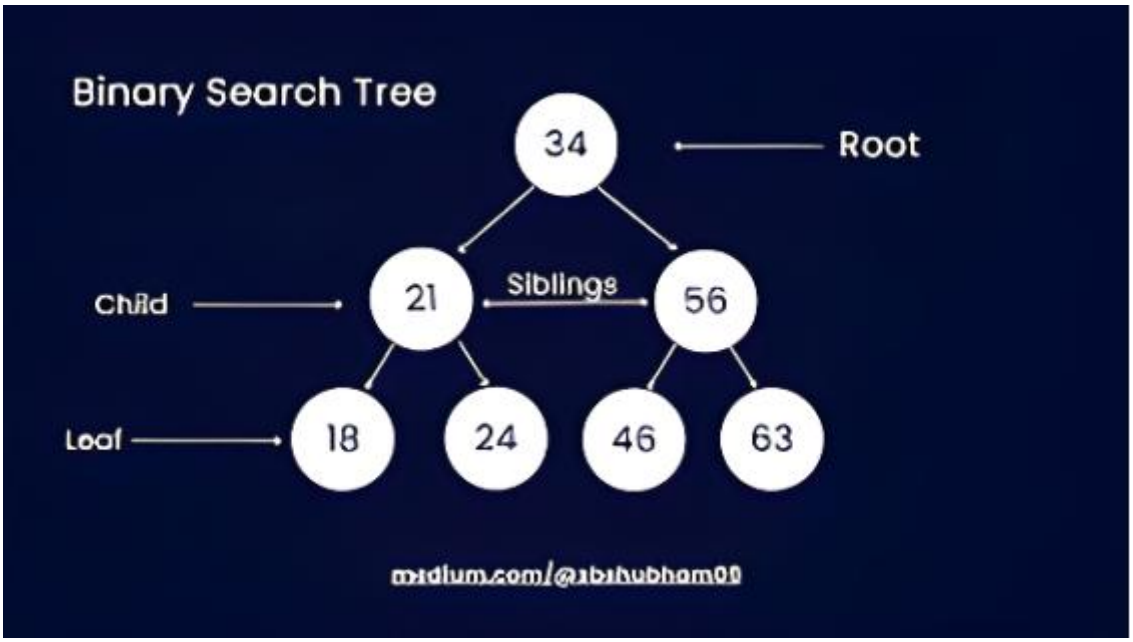


Figure 1: Components of Tree-Based Search Framework

The technical efficacy of the tree-based search framework is fundamentally dependent on the underlying data structures that manage the state-space. As illustrated in Figure 1, the framework consists of six distinct main components that combine together to generate a structured search process. In detail, the following description showcases each responsibility and function:

- Root: Serves as the origin of the search tree, representing the agent's initial starting state (x, y) before any exploration begins.
- Child: Represents a successor state generated by applying a valid movement action to its parent node.
- Siblings: Nodes that share the same parent, showing alternative path branches available at the same depth of the search.
- Leaf: Terminal nodes at the end of a branch that have not yet been expanded, collectively forming the "frontier" of the search.
- State: The core data stored within each node, containing the specific grid coordinates necessary to identify the agent's position.
- Parent Link: The directed connection (arrows) that allows the system to backtrack from the goal node to the root to reconstruct the final solution path.

Thus, my team believes that by having a deep understanding of these components, the agent guarantees that the transition from simple grid coordinates to complex search logic remains robust and results in precise actions.

2.1.3 The Mechanism Workflow

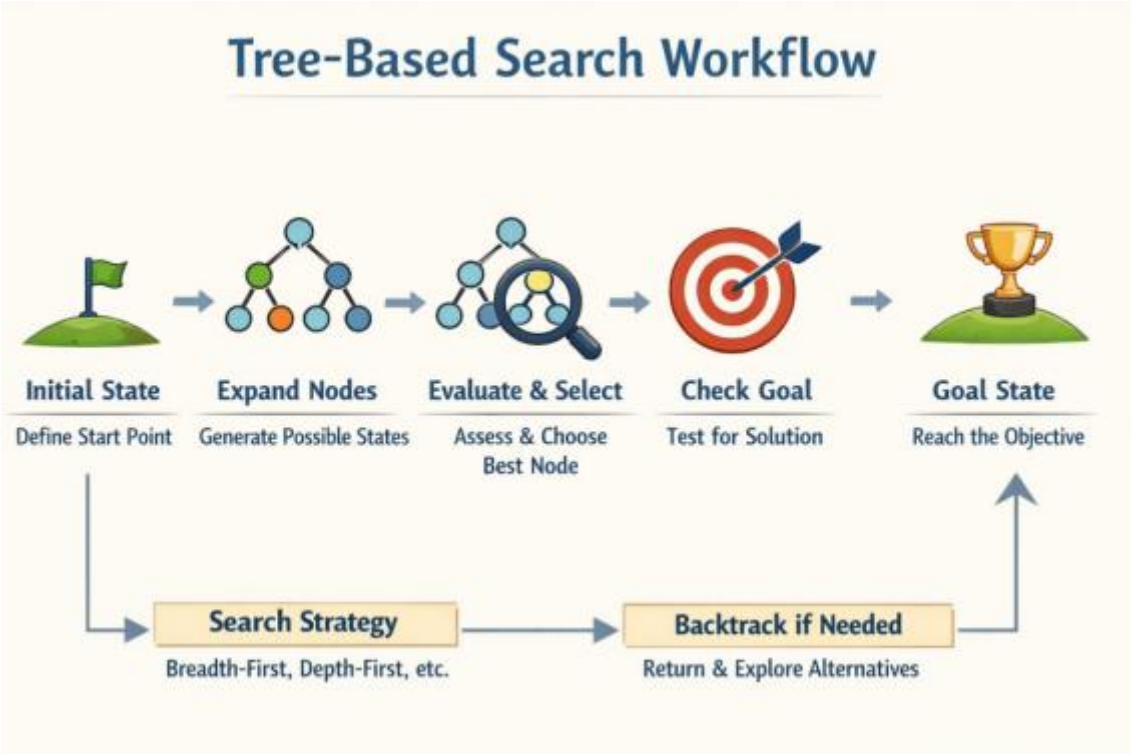


Figure 2: Tree-Based Search Workflow

The operational logic of the search framework follows a systematic and iterative process with strict rules. Therefore, our team can confidently program an agent that operates with high reliability and precision by adhering to these stages. Particularly, the workflow is executed through the following steps:

- Initialization Rule: Firstly, the process begins with encapsulating the starting coordinates into a Root node. Therefore, the node is the first entry in the Frontier, acting as the primary queue for pending exploration.
- Selection and Goal Test Rule: In each iteration, the agent selects a node from the Frontier phase based on the predefined priority of the chosen algorithm. Additionally, the rule in this stage is to perform a Goal Test that compares the current node's State with the target coordinates.
- Expansion Rule: If the goal test fails, the agent expands the current node by identifying all valid adjacent moves. Besides, each new move generates a Child node which is strictly linked to its Parent to maintain path integrity.
- Backtracking Rule: Once the Goal Test is successful, the agent ignores all other nodes in the Frontier and follows the Parent Links back to the Root. This ensures a reliable and easy-to-understand path is reconstructed for the agent to follow.

Through the structured workflow, our team generates a reliable foundation that ensures the search logic remains robust and deterministic. By following the stages strictly, we can effectively evaluate different algorithms and choose the best one to prioritize the Frontier to achieve specific performance goals. Consequently, this consistency allows the agent to maintain high precision regardless of the complexity of the environment.

2.1.4 Practical Application

Nowadays, the wide application of tree-based search frameworks spans across various perspectives, proving that a structured state-space is essential for modern automation. By integrating a robust search logic, the framework can be adapted to solve various real-world challenges:

- Autonomous Robotics and Navigation: The most stunning application is in autonomous mobile robots, particularly warehouse AGVs (Automated Guided Vehicles). In addition, the framework allows these robots to calculate the most efficient path from a starting position to a target shelf while avoiding static obstacles.
- Logistics and Route Optimization: In graph-based environments—similar to the data provided in PathFinder-test.txt —the algorithm can be used to optimize delivery routes between cities (nodes) connected by highways (edges), ensuring that goods are transported using the least amount of resources or time.
- Network Data Routing: The systematic movement from a Root node to a Goal is fundamental in routing data packets across the internet to ensure information reaches its destination via the most reliable path.
- Artificial Intelligence in Gaming: Tree search is the backbone of NPC (Non-Player Character) movement, allowing agents to navigate complex virtual environments deterministically.

By mastering fundamental theories and knowledge, our team can transform them into practical solutions that require high precision and reliability in dynamic environments. Notably, the bridge between theory and practice ensures that our agent is not only just a classroom project but also a functional prototype capable of navigating complex data structures like the one provided in our test scenario.

2.1.5 Benefits and Drawbacks

Although the powerful tree-based search is widely applied in various aspects of modern automation and computational intelligence, the framework still has invisible following potential benefits and drawbacks:

Benefits:

- Systematic Problem Solving: The framework provides a disciplined and logical processing workflow, preventing the agent from wandering aimlessly and ensuring every movement is goal-oriented.
- High Verifiability and Transparency: By utilizing available Python libraries include matplotlib and networkx, our team can draw data onto a Cartesian plane that allows for instant visual confirmation of node coordinates and ensures the digital model is accurate.
- Deterministic Reliability: By following strictly rules in node expansion and goal testing, the framework guarantees consistent and predictable results, which are vital for maintaining System Integrity.
- Maintainable Structure: The modular design categorizes nodes into Roots, Children, and Leaves, which makes the codebase highly accessible for debugging, fixing, and future development.

Drawbacks:

- Computational Resource Consumption: As the state-space grows, the number of nodes stored in the frontier can increase dramatically that leads to high memory usage.
- Dependency on Data Integrity: The success of the search is completely dependent on the quality of the input, and any error in the "PathFinder-test.txt" file would result in an invalid simulation.
- Risk of Non-Optimal Solutions: Depending on the specific strategy (like DFS), the framework may prioritize reaching the goal quickly over finding the shortest route.

Thus, tree-based search provides a powerful and systematic framework for the Pathfinding Problem: deterministic, modular, and heuristic-friendly. However, careful consideration of memory, algorithmic suitability, and performance bottlenecks are crucial criterion for achieving efficiency and scalability in complex environments. Furthermore, our team can design a system that is correct, fast, and scalable for various scenarios by understanding both advantages and disadvantages.

2.2 Uninformed Search Strategy

First and foremost, the tree-based search framework is divided into two distinct components: Uninformed Search and Informed Search

- Uninformed Search (Blind Search): The category of search algorithms that explore a problem space without using any additional knowledge or heuristics about how close a state is to the goal.
- Informed Search (Heuristic Search): The strategy utilizes a Heuristic function (h(n))to estimate the distance to the destination. By "aiming" toward the goal, algorithms like A* and GBFS significantly optimize the Exploration Ratio.

2.2.1 Depth-First Search (DFS)

Firstly, Depth-First Search (DFS) prioritizes exploring the deepest possible nodes in the search tree before backtracking to explore alternative branches. In addition, the algorithm utilizes the LIFO (Last-In, First-Out) technique, which indicates that the most recently discovered node is the first to be expanded. Particularly, the search uses the stack to manage the frontier, enabling the agent to seek vertically into the state space. Then, the agent pops the last node from the stack to backtrack and explore the next available branch when it reaches a leaf node or a dead end.

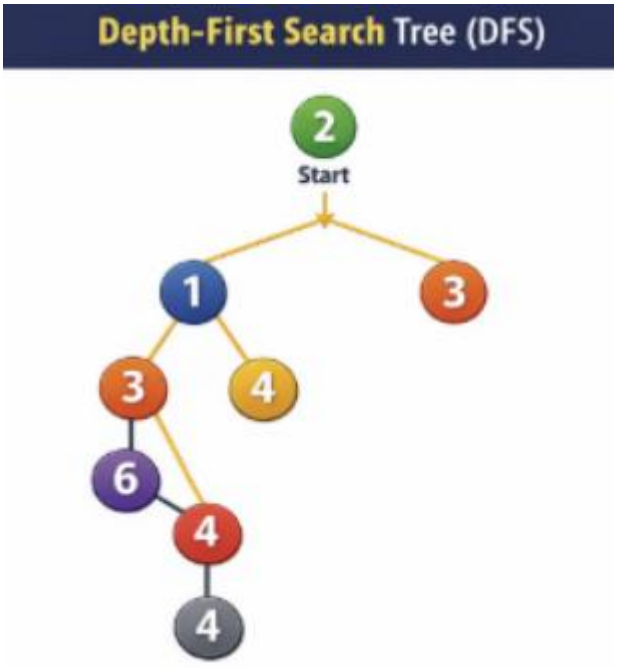


Figure 3: Visual Representation of DFS Node Expansion Order in a Search Tree

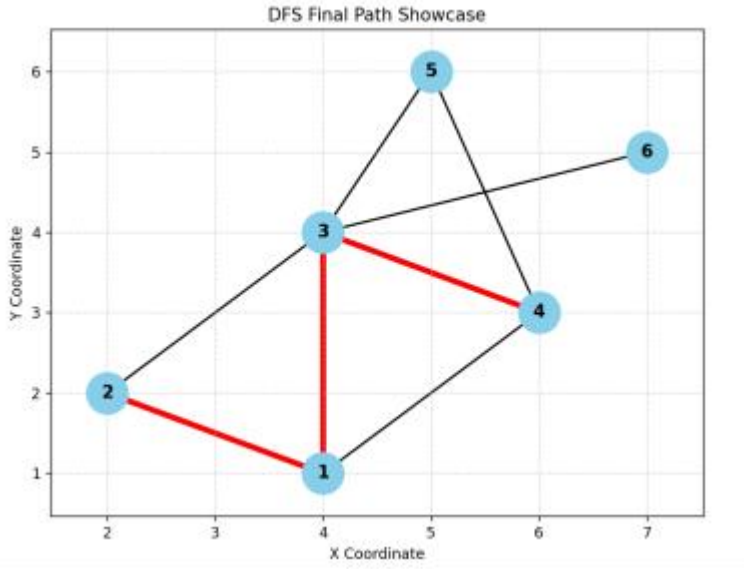


Figure 4: DFS Path Trajectory and Final Result on Coordinate

```python
#--Depth-First Search (DFS) Implementation--#
def dfs_algorithm(graph, start, goals):
    frontier = [(start, [start])] # Stack storage (current node, paths have visited)
    visited = set()               # Set for nodes to avoid loop back

    while frontier:
        current_node, path = frontier.pop() # LIFO: pop from the end of the list (stack behavior)

        if current_node in goals:
            return path # Return the path to the goal if found

        if current_node not in visited:
            visited.add(current_node)
            neighbors = list(graph.neighbors(current_node))
            # Sort neighbors in reverse order to ensure consistent traversal order (optional)
            for neighbor in sorted(neighbors, reverse=True):
                if neighbor not in visited:
                    frontier.append((neighbor, path + [neighbor]))
    return None
```

Figure 5: Python Implementation of the Depth-First Search (DFS) Algorithm using a Stack-based Approach.

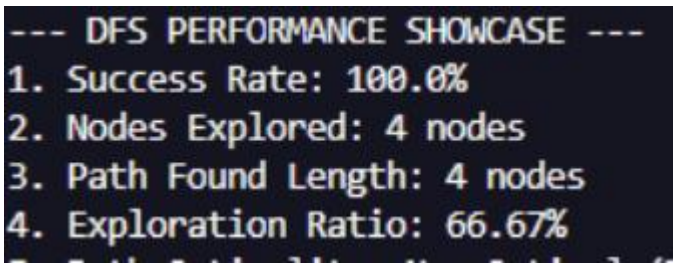


Figure 6: DFS Performance Metrics Summary

Thus, my team decides to integrate Python coding to not only gain a deep understanding of the algorithm's mechanics but also to evaluate the efficiency of the search strategy based on the performance showcase table. Besides that, visualizing the node on the Cartesian (Oxy) plane allows us to check the actual path trajectory and verify the accuracy of the algorithm's expansion logic against the theoretical model. Specifically, 66.67% of exploration ratio showcases the proportion of the state space

that the algorithm had to visit before identifying the goal, and it's extremely crucial when comparing DFS with other uninformed search methods like BFS or UCS.

2.2.2 Uniform-Cost Search (UCS)

While the DFS algorithm explores based on depth, the Uniform-Cost Search (UCS) strategy prioritizes nodes according to their cumulative path cost from the root. Specifically, the innovative strategy expands the node n with the lowest path cost g(n) first, guaranteeing that the algorithm systematically evaluates the cheapest available options. As a result, this approach makes the agent both complete and optimal even in graphs with varying edge costs. In regard to data structure, UCS utilizes a Priority Queue (frontier) which always keeps the node with the minimum cumulative cost at the front. Furthermore, the search radiates outward in "contours" of increasing cost, ensuring no expensive path is pursued until all more affordable alternatives have been exhausted.

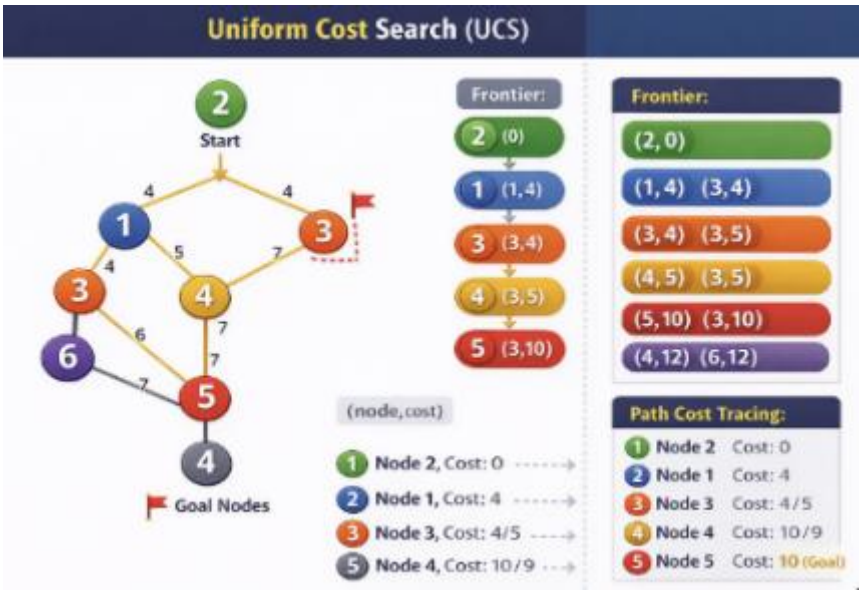


Figure 7: UCS Diagram Workflow

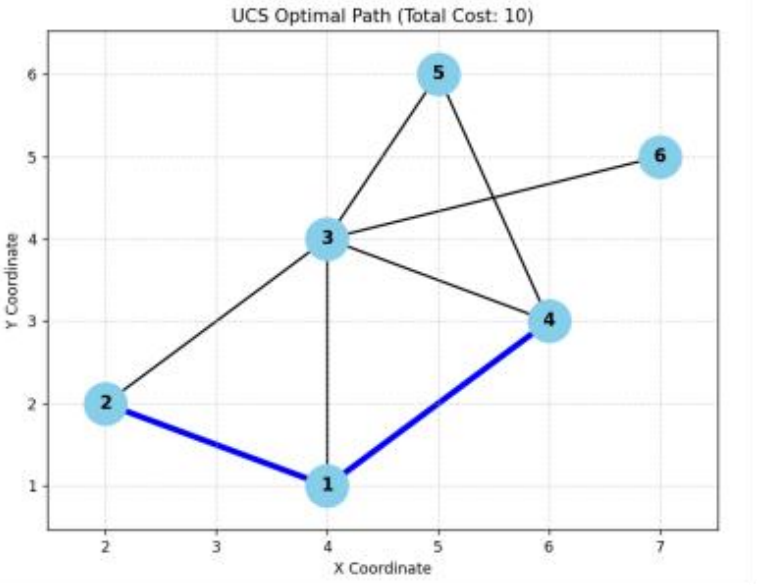
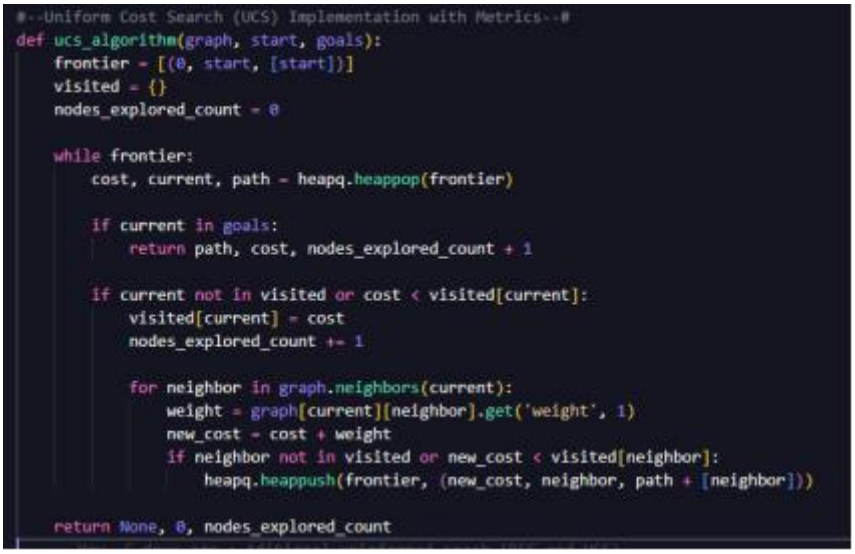


Figure 8: Pathfinding result in Oxy plane



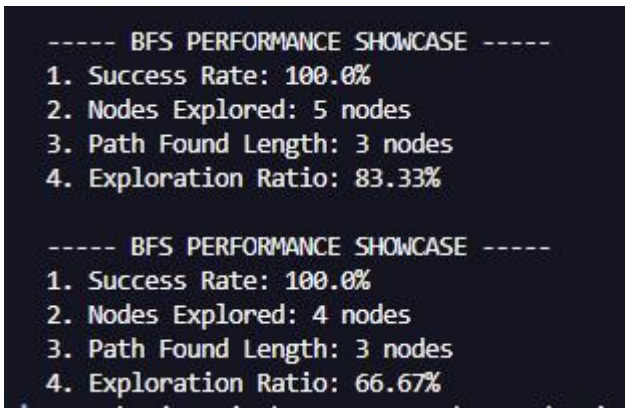
```python
#--Uniform Cost Search (UCS) Implementation with Metrics--#
def ucs_algorithm(graph, start, goals):
    frontier = [(0, start, [start])]
    visited = {}
    nodes_explored_count = 0

    while frontier:
        cost, current, path = heapq.heappop(frontier)

        if current in goals:
            return path, cost, nodes_explored_count + 1

        if current not in visited or cost < visited[current]:
            visited[current] = cost
            nodes_explored_count += 1

            for neighbor in graph.neighbors(current):
                weight = graph[current][neighbor].get('weight', 1)
                new_cost = cost + weight
                if neighbor not in visited or new_cost < visited[neighbor]:
                    heapq.heappush(frontier, (new_cost, neighbor, path + [neighbor]))

    return None, 0, nodes_explored_count
```

Figure 9: UCS Code Implementation in Python

It is clear that, though both node destinations in the pathfinding test are node 5 or 4, the UCS search demonstrates its superiority by consistently identifying the shortest and most cost-effective route. Particularly, the route result in Figure 8 takes only 2 paths to reach the end node instead of three paths in Figure 4. Consequently, this evidence proves that UCS is fundamentally more profound in achieving optimality than DFS, as it prioritizes total path cost over simple depth-first exploration. By systematically evaluating edge weights, the agent avoids the redundant movements observed in uninformed traversal, ultimately ensuring the most efficient resource allocation for the pathfinding task.

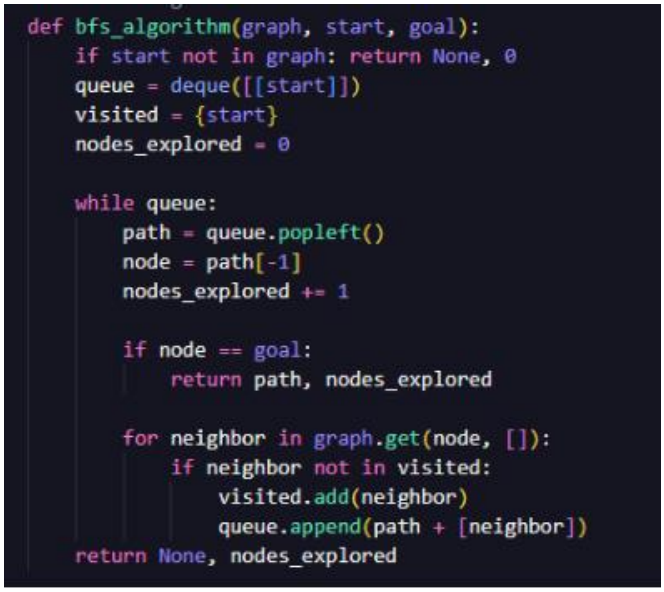
2.2.3 Breadth-First Search (BFS)

Finally, the Breadth-First Search (BFS) explores the search tree layer-by-layer, expanding all nodes at the current depth before moving to the next level. While the strategy shares the same goal of state space exploration as Depth-First Search (DFS), it utilizes the FIFO technique through a Queue structure to manage the frontier, which ensures the shallowest nodes are prioritized. As a result, the horizontal expansion allows the agent to guarantee the shortest path in terms of steps, contrasting with the vertical, LIFO-based approach of DFS.



```
----- BFS PERFORMANCE SHOWCASE -----
1. Success Rate: 100.0%
2. Nodes Explored: 5 nodes
3. Path Found Length: 3 nodes
4. Exploration Ratio: 83.33%

----- BFS PERFORMANCE SHOWCASE -----
1. Success Rate: 100.0%
2. Nodes Explored: 4 nodes
3. Path Found Length: 3 nodes
4. Exploration Ratio: 66.67%
```

Figure 10: BFS Performance Showcase Table



```python
def bfs_algorithm(graph, start, goal):
    if start not in graph: return None, 0
    queue = deque([[start]])
    visited = {start}
    nodes_explored = 0

    while queue:
        path = queue.popleft()
        node = path[-1]
        nodes_explored += 1

        if node == goal:
            return path, nodes_explored

        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(path + [neighbor])
    return None, nodes_explored
```

Figure 11: BFS Code Implementation in Python

Specifically, the BFS uninformed search figures out a stunning advantage by not only identifying both potential goal nodes (4 and 5) but also guaranteeing the discovery of the absolute shortest and fastest path in terms of edge count. As illustrated in Figure 11, the percentage of the Exploration Ratio is 83.33%, which indicates that the agent never misses a more efficient route, ultimately proving that BFS is the most reliable strategy for finding the optimal solution in unweighted environments where every step counts.

In conclusion, BFS stands out for its ability to identify the absolute shortest path in terms of steps by evaluating all potential goals like nodes 4 and 5 simultaneously, while DFS is memory-efficient, but risks suboptimal paths, and UCS guarantees the cheapest route based on cumulative cost. Consequently, this hands-on implementation not only bridges the gap between abstract logic and practical application but also provides a rigorous benchmark for selecting the most effective search strategy in real-world coordinate environments

2.3 Informed Search Strategy

The Informed Search Strategies uses problem-specific knowledge to guide the agent toward the goal more efficiently. Furthermore, this innovative approach is achieved by a Heuristic function h(n) which estimates the cost from the current node n to the nearest goal. Consequently, these searches can prioritize branches that appear more promising, significantly reducing the search space and the Exploration Ratio.

2.3.1 Greedy Best-First Search (GBFS)

The Greedy-Best First Search (GBFS) is an informed strategy that expands the node that is closest to the goal as estimated by a heuristic function (f(n) = h(n)). Moreover, GBFS acts greedily to reach the goal as quickly as possible by focusing solely on the remaining distance to the destination. In regard to the mechanism workflow, the search ignores the cost already spent (g(n)) and prioritizes nodes with the lowest h(n). Although it frequently results in a very low Exploration Ratio and high speed, it does not guarantee the shortest path (non-optimal) and can easily get stuck in local optima.

```python
def gbfs_algorithm(graph, positions, start, goal):
    if start not in graph or goal not in positions: return None, 0

    # Priority Queue stores: (h(n), current_node, path)
    # GBFS only cares about h(n) - the distance to the destination
    start_h = heuristic(positions[start], positions[goal])
    frontier = [(start_h, start, [start])]
    visited = set()
    nodes_explored = 0

    while frontier:
        h_val, current, path = heapq.heappop(frontier)

        if current in visited: continue
        visited.add(current)
        nodes_explored += 1

        if current == goal:
            return path, nodes_explored

        for neighbor in graph.get(current, []):
            if neighbor not in visited:
                h_neighbor = heuristic(positions[neighbor], positions[goal])
                heapq.heappush(frontier, (h_neighbor, neighbor, path + [neighbor]))

    return None, nodes_explored
```

Figure 12: GBFS Code Implementation in Python

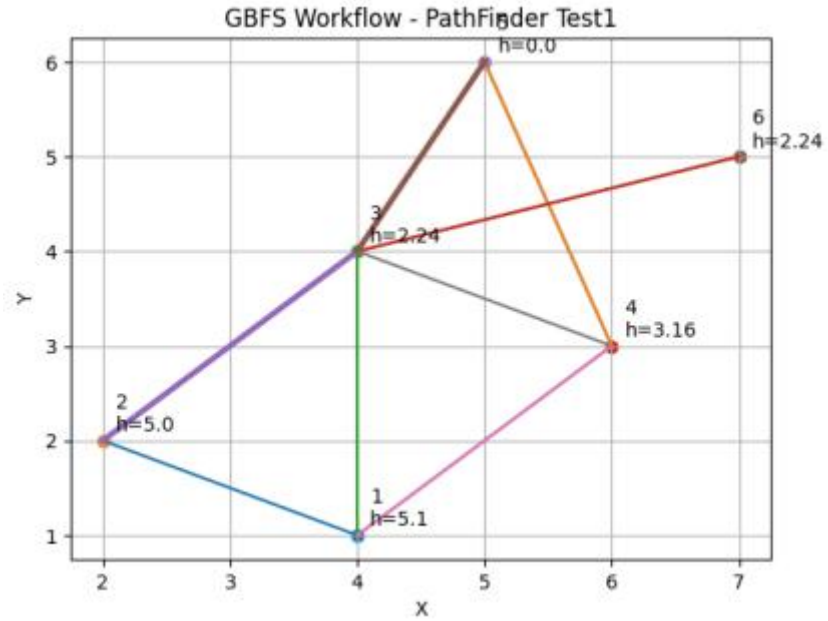


Figure 13: Node Expansion Order and Solution Path of Greedy Best-First Search (GBFS)

```
================================================
        --- GBFS PERFORMANCE SHOWCASE ---
================================================
1. Success Rate:          100%
2. Nodes Explored:        3 nodes
3. Path Length:           3 nodes
4. Exploration Ratio:     50.00%
5. Heuristic Used:        Euclidean Distance
6. Strategy:              Greedy (Best-First)
================================================
```

Figure 14: GBFS Performance Showcase Table

Therefore, the integration of the GBFS algorithm, as indicated in Figure 12, represents a significant shift from 'blind' exploration to informed navigation. In addition, the intelligent agent demonstrates a stunning efficiency that reaches a 50.00% Exploration Ratio in the performance showcase table by prioritizing nodes based on the Euclidean Distance heuristic. Specifically, Figure 13 highlights how leveraging problem-specific knowledge allows the search to bypass irrelevant states, focusing the computational effort directly toward the goal.

2.3.2 A* (A-star) Search

A-Star Search serves as the most powerful informed search because it effectively addresses the limitations of both UCS and GBFS strategies. Additionally, A* combines these two perspectives using the evaluation function, while UCS focuses only on the cost from the start, and GBFS focuses only on the estimated distance to the goal. In the following formula, g(n) represents the actual path cost from the starting node to node n, and h(n) is the heuristic estimate of the cost from n to the goal.

$$f(n) = h(n) + g(n)$$

```python
def a_star_algorithm(graph, positions, start, goal):
    if start not in graph or goal not in positions: return None, 0, 0

    start_h = heuristic(positions[start], positions[goal])
    frontier = [(start_h, 0, start, [start])]
    visited = {}
    nodes_explored = 0

    while frontier:
        f_val, g_cost, current, path = heapq.heappop(frontier)

        if current in visited and g_cost >= visited[current]: continue
        visited[current] = g_cost
        nodes_explored += 1

        if current == goal:
            return path, g_cost, nodes_explored

        for neighbor, weight in graph.get(current, []):
            new_g = g_cost + weight
            if neighbor not in visited or new_g < visited[neighbor]:
                h_neighbor = heuristic(positions[neighbor], positions[goal])
                new_f = new_g + h_neighbor
                heapq.heappush(frontier, (new_f, new_g, neighbor, path + [neighbor]))

    return None, 0, nodes_explored
```

Figure 15: A* Search Implementation in Python Code

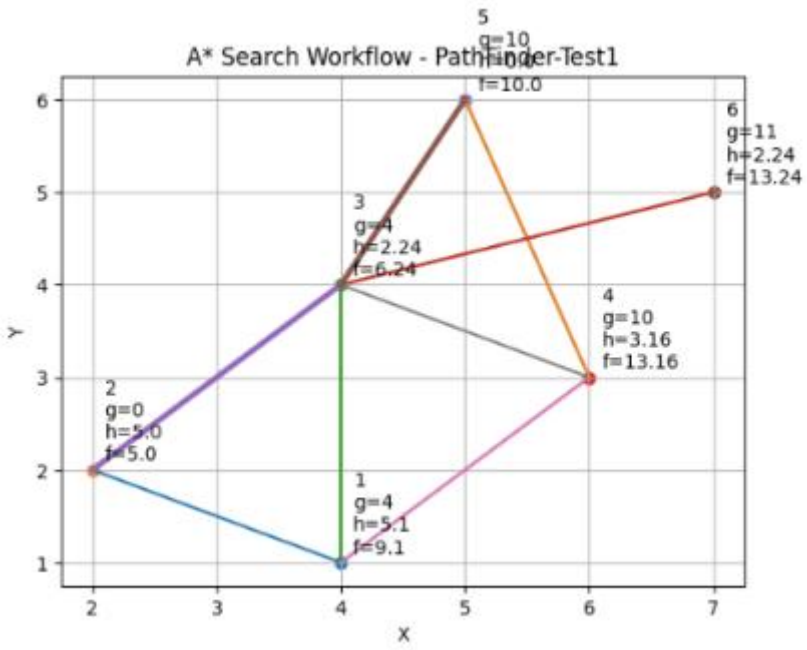


Figure 16: Node Expansion Order and Solution Path of A-Star Search

```
================================================
        --- A* PERFORMANCE SHOWCASE ---
================================================
1. Success Rate:          100.0%
2. Nodes Explored:        4 nodes
3. Exploration Ratio:     66.67%
4. Total Path Cost:       9
5. Optimality:            Guaranteed (Best Path)
================================================
```

Figure 17: A* Performance Showcase Table

Therefore, the primary advantage of A* Search is the ability to remain complete and optimal. In addition, the heuristic used is admissible, which means it never overestimates the actual cost to the goal. By integrating the cumulative cost g(n), A* avoids the "greedy" pitfalls of GBFSthat ensures the agent does not get stuck in local optima and always identifies the absolute shortest path.

2.3.3 Advanced Informed Search

Although A* and GBFS provide a solid foundation for coordinate-based pathfinding, large-scale or memory-constrained environments often require more specialized approaches. In detail, our team researches and explores the following two advanced options:

- Iterative Deepening A* (IDA*): A memory-efficient variant that combines the depth-first search (DFS) strategy with the heuristic guidance of A*. Additionally, it retains the optimality of A* while requiring significantly less memory by using a series of DFS searches with an f(n) limit.
- Beam Search: An optimization of the Best-First Search, which limits the number of nodes stored in the frontier to a fixed value k known as the Beam Width. As a result, the stunning strategy drastically reduces the Exploration Ratio and computational time, though it may sacrifice optimality in complex state spaces.

In summary, the transition from uninformed to informed search strategies illustrates a dramatic evolution in the efficiency of the agent's navigation. By coding and comparing strategies, our team observes a complete picture of how problem-specific knowledge transforms pathfinding. In addition, the application of informed search strategies particularly allows the agent to move beyond simple brute-force exploration. As demonstrated by our results, leveraging these strategies enables the system to bypass irrelevant paths, significantly reducing the Exploration Ratio from 83.33% in BFS to as low as 50.00% in GBFS and 66.67% in A*. Thus, the practical application proves that informed search is not just a theoretical improvement but a vital necessity for building intelligent agents that can navigate complex environments in real-world environments with both speed and precision.

Part C: Custom Search

During the research and analysis phase, our team figured out that the Beam Search strategy is completely superior to other search algorithms in terms of resource management and memory complexity. By restricting the number of active candidates at each level to a fixed Beam Width (k), this profound search prevents the exponential memory growth typically seen in A* or UCS. As a result, our analysis indicates that Beam Search is an outstanding choice for real-world real-time systems where speed and memory constraints are more critical than absolute path optimality.

Strategy	Optimality	Memory Usage	Frontier Size	Best For
UCS	Guaranteed	High	Unlimited	Shortest path (no heuristic)
GBFS	No	Moderate	Unlimited	Fast navigation (heuristic only)
A*	Guaranteed	High	Unlimited	Balanced efficiency and accuracy
IDA*	Guaranteed	Very Low	O (d)	Memory-constrained systems
Beam	No	Constant	Fixed (k)	Large-scale/Real-time AI

```python
def beam_search(graph, positions, start, goal, k=2):
    beam = [(heuristic(start, goal, positions), start, [start], 0)]
    nodes_explored = 0

    while beam:
        candidates = []
        nodes_explored += len(beam)

        for _, current, path, g_cost in beam:
            if current == goal:
                return path, nodes_explored, g_cost
            #--Node Neighbors--#
            for neighbor, weight in graph.get(current, []):
                if neighbor not in path:
                    new_path = path + [neighbor]
                    new_g = g_cost + weight
                    h = heuristic(neighbor, goal, positions)
                    candidates.append((h, neighbor, new_path, new_g))
        #--Beam Width--#
        beam = heapq.nsmallest(k, candidates, key=lambda x: x[0])
    return None, nodes_explored, 0
```

Figure 18: Beam Search Implementation in Python Code

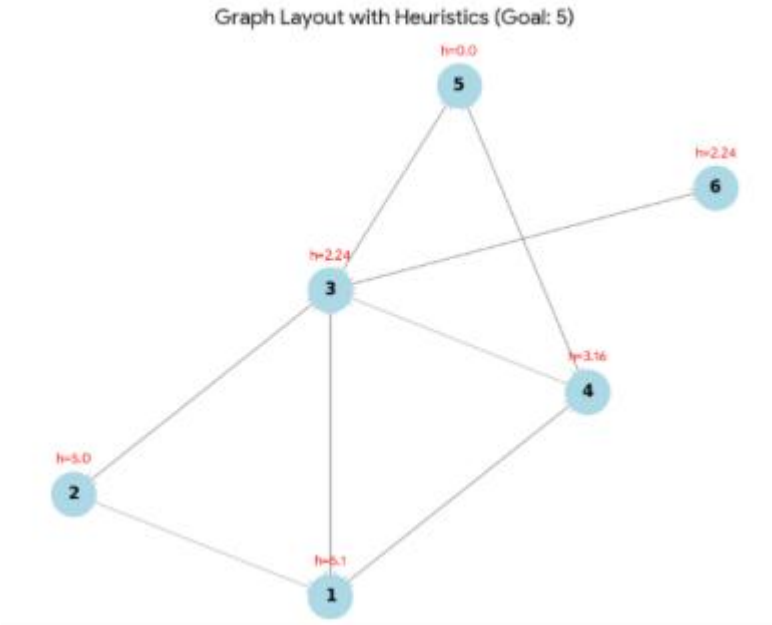


Figure 19: Graph Layout of Beam Search

```
--- RESULTS (Beam Width k=2) ---
1. Success Rate:      100.0%
2. Nodes Explored:    5 nodes
3. Exploration Ratio: 83.33%
4. Total Path Cost:   10.00
5. Optimality:        Limited by k=2
Final Path: 2 -> 3 -> 5
```

Figure 20: Beam Search Results

As my team expected, Beam Search not only showcases exactly the result of the path to the goal node but also demonstrates a highly controlled expansion process, which is shown in Figure 20. Consequently, the algorithm prioritizes the most promising candidates based on their heuristic values (h) at each level of the search tree by utilizing a Beam Width of k=2. Particularly, the parameter k=2 serves as a specialized filter that strictly manages the search frontier at each step, that provide following core benefits:

- Frontier Limitation: At every level of the search tree, the strategy only retains the top 2 most promising candidates with the lowest heuristic values (h) instead of keeping all neighboring nodes in memory like A*.
- Memory Efficiency: By setting (k=2), the memory footprint remains constant and predictable, preventing the exponential data explosion typically encountered in traditional search algorithms when navigating complex maps.
- Pruning Mechanism: Any node that falls outside the "top 2" is immediately discarded (pruned), which allows the algorithm to focus all computational power on the most direct paths toward the goal (Node 5) and ensures rapid execution.

In summary, the intelligent agent represents a profound strategic trade-off with k=2 and prioritizes speed and memory conservation by narrowing the search focus to only the two most likely paths at any given time

Part D: Conclusion

In conclusion, the completion of Assignment 2a offers our a profound opportunity for our team to bridge the gap between abstract AI theories and functional software engineering. By tackling the Route Finding Problem, the team successfully navigated the complexities of state-space exploration, which transforms fundamental search principles into a robust Python-based application.

- Integration of Theory and Practice: During workflow, the team decided to integrate the theoretical frameworks discussed in lectures with practical coding implementation, allowing team members to gain a deeper understanding of how abstract concepts, including frontier management and heuristic evaluation, operate in a real-world environment.
- Advanced Heuristics: The implementation of A*, IDA*, and Beam Search proved that informed search strategies are significantly more efficient than blind exploration, especially when dealing with specific constraints like memory limits.
- System Determinism: By adhering to strict tie-breaking rules, the team ensured that the agent's behavior remains entirely deterministic and predictable, which is vital for real-world robotics and logistics.

Thus, the assignment acts as a testament to effective teamwork. Throughout the workflow, the relationship between team members became more collaborative and synchronized, fostering an environment where algorithmic analysis and system development could thrive. Furthermore, we were able to deliver a robust application that successfully navigates the complexities of the Route Finding Problem by aligning our individual strengths. Therefore, this experience has not only sharpened our technical skills in AI development but also reinforced the importance of systematic, collective problem-solving in the face of modern computational challenges

Part E: Acknowledgements

During the writing process, our team has used a combination of academic research from prestigious organizations and advanced AI assistance. In detail, the following lists are:

l Verification: After completely finishing, my team reviewed and adjusted every section to ensure that the AI's assistance remained secondary to our research and intellectual direction.

l GitHub Repository: All materials, including code files and crucial documents, are hosted on our GitHub which provides a transparent audit trail of our development process and collaborative efforts.

l Google Docs: The platform served as our primary platform for real-time collaboration, enabling the team to synchronize research findings, draft the comparative analysis between Informed and Uninformed search, and refine the final documentation seamlessly.

Part F: References

LaValle, & S. M. (2006). *Planning Algorithms / Motion Planning*. Uiuc.edu. http://planning.cs.uiuc.edu/

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961