Cover sheet for submission of work for assessment

UNIT DETAILS

Unit name	Introduction to Artificial Intelligence			Class day/time	Tue 8:30
Unit code	COS30019	Assignment no.	2A	Due date	13 April
Name of lecturer/teacher	Duy Khoa Pham				
Tutor/marker's name	Duy Khoa Pham				

Office use only

Faculty or school date stamp

STUDENT(S)

	Family Name(s)	Given Name(s)	Student ID Number(s)
(1)	Taslim	Abrar	103825785
(2)	Jahir	Dewan Md Amir	104071152
(3)	Haider	Abdur Rehman	104654906
(4)	Saad	Md Mudabbirul Islam	105281389
(5)			
(6)			

DECLARATION AND STATEMENT OF AUTHORSHIP

1. I/we have not impersonated, or allowed myself/ourselves to be impersonated by any person for the purposes of this assessment.
2. This assessment is my/our original work and no part of it has been copied from any other source except where due acknowledgement is made.
3. No part of this assessment has been written for me/us by any other person except where such collaboration has been authorised by the lecturer/teacher concerned.
4. I/we have not previously submitted this work for this or any other course/unit.
5. I/we give permission for my/our assessment response to be reproduced, communicated, compared and archived for plagiarism detection, benchmarking or educational purposes.

I/we understand that:

6. Plagiarism is the presentation of the work, idea or creation of another person as though it is your own. It is a form of cheating and is a very serious academic offence that may lead to exclusion from the University. Plagiarised material can be drawn from, and presented in, written, graphic and visual form, including electronic data and oral presentations. Plagiarism occurs when the origin of the material used is not appropriately cited.

Student signature/s

I/we declare that I/we have read and understood the declaration and statement of authorship.

(1)	Abrar	(4)	Saad
(2)	Dewan	(5)	
(3)	Haider	(6)	

Further information relating to the penalties for plagiarism, which range from a formal caution to expulsion from the University is contained on the Current Students website at **www.swin.edu.au/student/**

Copies of this form can be downloaded from the Student Forms web page at **www.swinburne.edu.au/studentforms/**

Table of Contents

Statement of Contribution

Student Name	Contribution
Abrar Taslim	Designing the Custom Search Function, Designing the Graph Function, Report Writing (Section 1,3,6,7)
Abdur Rehman Haider	Designing the DFS and BFS search function, Report Writing (Section 2,3,6)
Dewan Md Amir Jahir	Designing the input data, parser and search function, Designing the Custom Search Function, Report Writing and Formatting (Section 4 and 5)

Md Mudabbirul Islam Saad	Designing the A*, DBFS, BDWA, Custom 2 search functions. Designing the entire test cases and implementation of the project.

The GitHub Repository of The Project

https://github.com/AmirAX17/Assignment2forAi.git

1. Instructions

1.1. Program Overview

The program executes searches for a path solution through the Route-Finding Problem by processing structured input files that build a graph for application of multiple search algorithms to find valid paths from origin to destination nodes. The program implements a highly modular design structure because each element serves one distinct purpose for code reuse and straightforward testing and design clarity.

The Input Parser module stands as the central feature of the program within input_parser.py. The input_parser module handles processing of the input_data.txt file by dividing its contents into four specific parts - Nodes, Edges, Origin, and Destinations [2]. The sections in this program contain vital information with Nodes displaying node coordinates ("1: (4,1)") and Edges showing node connections and their associated cost "(2,1): 4" while the Origin and Destinations portion identifies the start and end points of the search. The parser transforms the unprocessed text data into Python-ready data collections consisting of node dictionaries and edge dictionaries as well as origin parameters and destination lists which the program receives subsequently [3].

The Graph Module (graph.py) adopts parsed data through the Graph class implementation. It contains the node and edge elements while giving access to helper methods that return sorted lists of neighbor nodes. During search operations the sorting process plays a vital role since it enables equal nodes to consistently break ties based on their matching evaluation criteria [3]. This abstraction enables simpler search algorithm logic and independently operates the graph data structure from the search algorithms.

The search algorithms represent the main operational capabilities through their sets of search algorithm modules.

- The bfs_search.py and dfs_search.py modules represent uninformed search approaches to map exploration that does not involve problem-specific knowledge. BFS implements a queue structure that allows it to find shortest path lengths through edge counts while DFS applies a stack that provides memory efficiency at the cost of potentially finding longer paths [4].
- A* (astar_search.py) along with GBFS (gbfs_search.py) implement heuristic information (primarily based on Euclidean distance) that directs the search process by relating the accumulated cost to projected goal expenses [1].
- The search implementation contains a custom uninformed method called CUS1 which relies on IDDFS (iddfs_search.py) for execution. The IDDFS technique performs repeated search expansions while growing search depth which unites DFS memory efficiency with BFS structural planning [2].
- The search.py driver script serves as a front-end user selection tool which allows users to execute various search algorithms from one interface. The driver executes the process by accepting both search method (BFS, DFS, GBFS, AS, CUS1,CUS2) along with input file name through command-line arguments. The program obtains graph data from

the common input parser after which it instantiates Graph class and delegates program control to target search functionality. The driver shows search results through a consistent output which delivers information regarding reached target nodes along with generated node counts and entire route path from source to target. The program renders an optional friendly output by substituting node numbers with descriptive names that exist in the input document [4].

The project is designed to be executed from the command line. We can choose among various search algorithms by specifying the method as an argument. Follow these steps to run the program:

1. Open the Terminal in the Project Folder:
2. Navigate to the directory containing the project files.
3. Activate the Virtual Environment:
4. Run the Integrated Search Driver:

Use the following command format to execute a search:
python3 search.py input_data.txt <method>
Replace <method> with one of the available options:
BFS (Breadth-First Search)/DFS (Depth-First Search)/GBFS (Greedy Best-First Search)/AS (A* Search)/CUS1 (Custom Uninformed Search via IDDFS)/CUS2.
For Example, To run BFS:
python3 search.py input_data.txt BFS
The program will load the graph data from input_data.txt, execute the chosen search algorithm, and print the result. The output will include:

- The file name and chosen method.
- The goal node reached.
- The total number of nodes generated.
- The full path from the origin to the goal.
- An optional human-readable version of the path, if node names are provided [3].

1.2. Overall Workflow:

- The program conducts file reading operations on input_data.txt to extract nodes and edges together with origin and destinations parameters.
- The collected data gets arranged into a Graph object which functions as an easy traversal mechanism.
- Search Execution uses the selected algorithm between BFS, DFS, A* and GBFS and user-created custom searches according to user preferences.
- The program displays the solution composed of the goal node together with node count along with path information in a standardized manner.
- Flexible maintenance along with straightforward addition exists because the program design follows a modular approach. The project building process involves developing independent components which subsequently integrates them to enable effortless addition of new search algorithms and features like graphical interfaces to the system. The approach produces a complete investigation of different search approaches within a logical and solid overall framework.

2. Introduction

The Route-Finding Problem is a classical pathfinding task where the objective is to determine the most efficient route between a starting node and one or more destination nodes on a 2D

directed graph. Each node represents a specific location defined by its coordinates, while edges define directed connections between nodes with associated costs. In this assignment, we implemented a suite of tree-based search algorithms designed to solve this problem. The task was approached from both uninformed and informed perspectives. The following algorithms were implemented:
Breadth-First Search (BFS): An uninformed strategy that explores all neighbors at the current depth before moving deeper. Depth-First Search (DFS): An uninformed strategy that dives deep into one branch before backtracking.
Greedy Best-First Search (GBFS): An informed strategy that uses a heuristic (Euclidean distance) to guide exploration.
A Search (AS):* An informed strategy that combines the cost from the start and the estimated cost to the goal.
Iterative Deepening Depth-First Search (CUS1): A custom strategy combining DFS's memory efficiency with BFS's completeness by iteratively increasing depth limits.
Bidirectional Weighted A*: CUS2 implements a Bidirectional Weighted A* Search algorithm with dynamic weighting. The algorithm simultaneously explores from the origin and the destination(s) using two separate priority queues and gradually reduces a heuristic weight factor (epsilon) to refine the search as it progresses.
The goal was not only to find a valid route but to evaluate the trade-offs between completeness, optimality, memory use, and performance across different strategies.
These algorithms were written in Python and organized into modular files to support clarity, scalability, and ease of testing. The solution also includes a flexible input parsing system and a command-line interface that accepts different algorithm arguments for batch testing or interactive evaluation.

3. Features/Bugs/Missing

3.1. Implemented Features

Input Parsing and Graph Construction:
The software has a dedicated input parser module that extracts structured data from the input file. This file contains node positions, directional edges with their costs and the original node a destination nodes. The data is used to instantiate a graph object graph.py which offers methods to access neighbors in sorted order—ensuring deterministic and reproducible node expansion.
Breadth-First Search (BFS) (bfs_search.py):
This search explored all nodes at their current depth level before going to the next. This search uses a queue to make sure that level-order transversal and guarantees that the shortest path is taken according to edge count. Consistent expansion order using ascending Neighbor sorting is also ensure.
Depth-First Search (DFS) (dfs_search.py):
This search goes deep along each branch using a stock makes sure memory is efficiently used. Neighbors are processed in ascending order unlike bfs and this is done to maintain lexicographic consistency when a stack is being used. Even though this isnt the best in terms of path cost DFS is fast and space efficient for specific graph structures.
Greedy Best-First Search (GBFS) (gbfs_search.py):
This is a strategy that used Euclidean distance to go to the nearest destination as a heuristic. It expands nodes with the lowest heuristic values using a priority queue, favouring fast solutions, although not guaranteed to be optimal.

A Search (AS)* (astar_search.py):
This search combines the actual path cost and heuristic estimates to find the path with the least cost. Priority queun and g_scores trackers are used with mechanisms for consistent node expansion order. A* guarantees optimal solutions with admissible heuristics.
Customer Uninformed Search (CUS1 – IDDFS) (iddfs_search.py):
This search uses Iterative deepening depth-first search which combines the space efficiency of DFS with the completeness of BFS and this increases the search depth. This implementation limits depth and reconstructs the path using a came_from dictionary.
Customer informed Search (CUS2 – Bidirectional Weighted A*) (bdwa_search.py):
A bidirectional heuristic-based search is a search that simultaneous performs forward and reverse A* searches. This search model used dynamic epsilon-weighting to balance heuristic guidance and low costs to make a highly efficient search model and minimizes node expansions and moves the counts in large graphs.
Search Driver Execution
The search.py module allows uses to run any of the 6 algorithms through command-line execution. It accepts both the input file and method name as arguments, invokes the appropriate search module, and standardizes output to include the goal node reached, the number of nodes generated and the path from origin to destination.

3.2. Known Bugs and Issues

1. Strict File Format: The parser needs strict file formatting since any deviations could trigger errors to occur [2].
2. Security Concerns: The parsing method which uses eval() makes security concerns more likely because it trusts all input [5].
3. Scalability: The performance of search algorithms starts to decline while working with large-scale graph data especially when iterative deepening methods are employed [7].

3.3 Missing Elements and Future Enhancements

1. GUI Development: The application would benefit from having a visual interface that improves its user experience [8].

2. Enhanced Error Handling: A solution should include robust checks along with safer alternatives for parsing [5].
3. Performance Optimizations: The investigation of better tie-breaking techniques for enhancing the performance of large-scale graph operations needs to be pursued [6].
4. Extended Testing: A complete set of tests should be designed alongside benchmarks to make future performance improvements possible [7].

4. Testing

Testing of the implemented search algorithms occurred through 10 designed real-world scenario test cases that assessed both operational stability and performance output. The researchers developed ten practical scenario test cases representing problems in urban routing and emergency rescue and university campus directions. Our research includes descriptions of the test case development approach along with testing methods Also included are the obtained data and analysis of observed findings.

4.1. Test Case Design

The development of ten test case scenarios led to separate input files for each instance. Multiple test cases of varied complexity were chosen because they let uninformed as well as informed search algorithms show their effectiveness under multiple situations. The test cases include:

1. City Transportation Network (test_case1.txt):

Scenario: Finding the optimal route from downtown to the airport.
During test execution a traveler moves between Downtown Central Station and the Airport Terminal.
Constraints: Multiple route options with various road distances (in km).
The task requires finding the most efficient route through an involved network of city roads.

2. Emergency Response Network (test_case2.txt):

During emergencies the test examines how ambulances would be routed to accessible hospital facilities.
Medical responders search for the quickest route between an accident location that leads to either hospital facility.
Travel times are affected by road traffic conditions which should be considered as constraints (in minutes).
The critical need exists for making decisions regarding multiple locations when operating under time constraints.

3. Warehouse Robot Navigation System (test_case3.txt):

Scenario: Autonomous robot navigation in a warehouse.
The delivery robot travels from the Loading Bay toward the Delivery Pickup Point.
Constraints: A grid layout with fixed movement costs.
The system needs to achieve efficient path planning for environments resembling mazes.

4. Melbourne Metro Train Network (test_case4.txt):

Scenario: Finding an optimal train route from Flinders Street to Glenferrie Station.
A university student makes a public transport journey between central Melbourne and Swinburne University.
The system operates based on fixed train routes along scheduled stoppoints that require possible transfer connections.
The system needs to identify train routes swiftly between various transportation lines and their connecting intervals.

5. Melbourne Tram Network (test_case5.txt):

Scenario: Determining the optimal tram route from Melbourne CBD to St Kilda Beach.
The tourist needs to find the fastest travel path that extends between the downtown area and a leading tourist destination.
The study considers tram schedule information together with the time needed to travel between specified stops.
Users face difficulties when searching through the multiple routes available on the extensive tram system.

6. Melbourne University Campus Navigation (test_case6.txt):

A student needs to determine the most suitable path for walking between university structures.
A student needs to determine the fastest ways to reach different parts of the university campus.
Constraints: Walking times between buildings.
The route requires service to various endpoints at this time.

7. Melbourne CBD Bicycle Path Network (test_case7.txt):

Scenario: Planning a cycling route in Melbourne CBD.
A cycling individual makes a trip from Southern Cross Station to RMIT or Carlton Gardens.
The cycling times fluctuate according to the selected terrain and distance of travel.

The main challenge consists of finding the fastest but also easy-to-manage path.

8. Melbourne Emergency Services Response (test_case8.txt):

The urban region of Melbourne requires route planning for fire department emergency responses.
Existing emergency response is managed by Eastern Hill Fire Station through its fire trucks.
Constraints: Traffic and time-critical response requirements.
The selected task requires finding the quickest routing approach for reaching various emergency destinations.

9. Melbourne Food Delivery Route (test_case9.txt):

Naval forces design delivery routes that serve food drivers in their excursions.
A driver needs to establish the fastest transportation path that leads from Lygon Street eateries to their end points.
High priority is given to delivery route travel durations between destination points.
The selection of efficient transportation paths which must be completed within restricted delivery times becomes the primary difficulty.

10. Melbourne Tourism Route (test_case10.txt):

Scenario: Determining an optimal walking tour for tourists.
The tourist designs a traveling path which includes significant Melbourne sites.
Different walking times exist between the attractions.
Tourists need to prioritize their must-see attractions so they can establish an optimal touring arrangement which includes numerous locations.

4.2. Testing Methodology

The entire set of implemented search algorithms including BFS, DFS, A* along with GBFS, CUS1, CUS2 received test cases for execution. The testing process included these procedure elements:

1. Setup: Our system received load orders of corresponding input files during each testing cycle. The input parser applied several processes to read various files which produced information about nodes and edges together with origin data and destination data. The retrieved data from the input files enabled the formation of a Graph object which was subsequently given to each search algorithm for execution.
2. Execution: The search.py program allowed us to execute each algorithm one after the other. A set of specific metrics was measured for every run.

- The goal node reached.
- The complete number of nodes which the search procedure created.
- The full path from the origin to the goal.
- Software execution time was measured through the built-in Python tools used for specific tests.

3. Validation: A manual check confirmed that the output paths maintained proper accuracy and maintained consistent results. We documented all inconsistencies related to algorithm performance through differences in path length or node generation data for subsequent evaluation.
4. Performance Benchmarking: We performed benchmarking of these algorithms using different graph sizes that resulted from manipulating the complexity of input test files. The experimentation showed the behavior of each algorithm when dealing with additional nodes and edges.

4.3. Analysis of Testing

The City Transportation Network test identified BFS as effective in reaching the shortest edge-based route while displaying amplified memory usage across growing graph levels. The DFS method performed quicker under basic circumstances yet it did not always identify the most efficient route. A* together with GBFS exhibited shorter time to solve and required fewer node expansions because of their Euclidean heuristic adjustment capabilities during scenarios that presented defined geographic patterns.

Emergency Response Network gained its best performance by using heuristic-based approaches that selected nodes which produced rapid solutions. The accuracy of BFS led to longer execution times versus both GBFS and A* because these methods efficiently managed the sparse emergency network topology.

The Warehouse Robot Navigation System test demonstrated that IDDFS proved effective for CUS1. CUS1 continued to explore nodes multiple times because of depth increments but it managed memory usage in check and found safe pathways in complex environments. The execution duration took longer than A* and the straightforward search methods so it was noticeable during the test.

When testing occurred on the Melbourne Metro Train and Tram Networks the selection of heuristics became the key factor. The GBFS search protocol reached solutions quickly in certain cases but A* maintained its ability to find the best possible routes no matter the number of transfer points and connection points existed. The two custom algorithms CUS1 and CUS2 used in campus navigation and bicycle path network scenarios demonstrated successful execution of memory usage reduction and minimal movement requirements. The route optimization performed by CUS2 depended heavily on transition reduction because each change required a permanent cost penalty.

For emergency services and food delivery paths with strict deadlines the informed route-planning techniques (A* and GBFS) produced superior outcomes than uninformed methods regarding performance along with efficiency measures. CUS2 achieved faster search times through its bidirectional search method which worked from both starting and ending points.

5. Insights

5.1. *Comparing Search Algorithms (Performance Analysis)*

1. Memory Efficiency:

- When processing wide graphs BFS stores every node with the current depth in a queue thus causing memory utilization to become unreasonably high [2].
- The data structure DFS saves memory due to its storage method that uses one path between the root and leaf nodes [6].
- Through proper heuristic methods (A and GBFS) selection the open set maintains openness but reduces the stored node count effectively [1].
- Custom Uninformed Search based on IDDFS implements DFS memory efficiency while systematically expanding the search to maintain reasonable memory requirements during deeper evaluations [8].

2. Optimality and Accuracy:

- The shortest path detection by BFS depends on edge number yet it does not necessarily lead to minimal cost paths when edge weight values change [2].
- DFS fails to discover the highest-quality path because it enters extended paths before checking alternative steps [2].

- A* guarantees to find the optimal and most economical solution for any problem when using admissible heuristic functions [1].
- Because GBFS uses a greedy approach it will discover solutions rapidly yet may overlook the path with maximum potential [1].
- CUS1 uses depth-incremental methods to discover solutions while achieving complete results with optimal resource management [8].

3. Computational Speed: Our performance evaluation depended mainly on the speed parameter of all algorithms examined.

- When BFS investigates each level through complete exploration of its nodes before moving forward it ensures solution optimality by selecting the shortest edge paths yet it sacrifices performance speed on extended network graphs. Our experiments showed excessive search times during runtime because the branching factor expansion influenced BFS performance negatively [2].
- DFS demonstrates faster execution in smaller graphs because it follows a depth-based expansion strategy. The performance capability of this algorithm depends directly on how its nodes are expanded for exploration. DFS usually faces severe delays during execution because it explores extensive suboptimal paths before returning to previous branches when the shortest path has an obscure starting point.
- The heuristic function constitutes the principal determinant of resulting performance levels between A* and GBFS. The use of accumulated cost and heuristic ($f(n)=g(n)+h(n)$) within A* achieves efficient node expansion resulting in quicker optimal solution discovery provided that the heuristic stays admissible and consists [1]. In real-world implementations of GBFS the procedure operates based on heuristic values only yet there can be missed opportunities to find better solutions through this optimization method. The performance graphs show that A* together with GBFS reaches faster speeds than BFS and DFS when the heuristic provides accurate remaining cost predictions.
- The IDDFS approach of CUS1 needs additional computational resources because it executes repeated DFS operations with expanding depth limits. The repeated iterations within this method cause additional computation time even though they eventually lead to finding solutions with satisfactory memory profiles. Such applications usually require specific memory optimization measures even though the performance reduction poses a dilemma.
- The CUS2 software system performs a bidirectional weighted A* algorithm with dynamic weighting as it switches between forward and backward search moves. When both forward and backward search operate in tandem with a system that updates heuristic weights the resulting path-finding process becomes more efficient because it expands fewer nodes. Testing confirms that managing two search fronts at once alongside dynamic weight updates boosts per-iteration computing work yet leads to competitive runtime alongside better node generation when move count reduction remains the main objective [5].

5.2. Trade-offs Between Uninformed and Informed Methods

Search methods without specific supplementary information (BFS, DFS, CUS1) traverse the search area without heuristic guidance. The simplicity of these approaches leads to behavior patterns which can be anticipated sometimes. The exploration of numerous unnecessary nodes stems from uninformed methods when operating on extensive graphs which subsequently enhances computational processing time and memory requirements [2]. The search direction for A* and GBFS moves toward the goal through the application of heuristics. Heuristic quality determines how much speed and node expansion reduction occurs while employing this

strategy. Choosing a suboptimal heuristic will make informed methods search worse than uninformed methods [1]. The selection between uninformed methods offering dependable yet basic performance and informed techniques allowing effective and possibly optimal results creates an operational trade-off.

5.3. Impact of Custom Strategies (CUS1 and CUS2)

- CUS1 (Custom Uninformed Search): CUS1 employs Iterative Deepening Depth-First Search (IDDFS) to merge the memory efficiency of DFS with the systematic, breadth-like exploration of BFS. This approach incrementally increases the search depth, ensuring that the algorithm finds a solution without consuming excessive memory—even when the optimal depth is unknown [8].
- CUS2 (Custom Informed Search): CUS2 is designed to minimize the number of moves rather than the aggregate cost. By assigning a uniform cost to each step, this method redefines the optimization objective to focus on reducing transitions. This alternative strategy is particularly effective in scenarios where minimizing the number of moves yields better practical performance than merely lowering the cumulative weight of a path [5].

6. Research

6.1. External Resources and Literature Reviewed

We consulted foundational texts and online resources to guide our work. Hart et al. [1] provided essential theory for heuristic cost determination, while Russell and Norvig [2] and Cormen et al. [6] offered comprehensive insights into search algorithms. Practical guidance from GeeksforGeeks [3] and Real Python [4] helped refine our implementation, and Pearl [8] along with Poole and Mackworth [5] informed our strategies on heuristics and algorithm efficiency.

6.2. Enhancements Derived from Research

Research influenced key improvements in our project. The work of Hart et al. [1] and Pearl [8] helped us refine the heuristics for A* and GBFS, whereas concepts from Russell and Norvig [2] and Cormen et al. [6] guided the design of our custom uninformed search (CUS1). Online resources [3], [4] contributed practical coding techniques, and advice from Poole and Mackworth [5] plus Knuth [7] aided in optimizing tie-breaking and performance.

6.3. Discussion of Alternative Approaches

Alternative approaches, such as bidirectional search and the use of advanced data structures like Fibonacci heaps [7], were considered to enhance efficiency. Further exploration of adaptive heuristics and dynamic tie-breaking mechanisms suggested by Pearl [8] and Cormen et al. [6] could yield additional performance gains in large-scale graph scenarios.

7. Conclusion

Our project successfully implemented a variety of search algorithms to solve the Route-Finding Problem. Uninformed methods like BFS and DFS, while straightforward, often require significant memory and processing time on large graphs [2]. In contrast, informed methods such as A* and GBFS, leveraging heuristics, provide more efficient and optimal solutions [1]. Our custom searches, CUS1 and CUS2, further demonstrate that innovative approaches using

iterative deepening and move minimization techniques—can effectively balance memory efficiency, speed, and accuracy.

Based on our findings, A* consistently offers a strong balance between optimality and efficiency when paired with a well-designed heuristic. However, in contexts where memory is limited or the search depth is unknown, methods like CUS1 offer a robust alternative. Future work should focus on integrating a graphical user interface, enhancing error handling, and further optimizing heuristic functions and tie-breaking strategies.

In summary, while each algorithm has its strengths and weaknesses, the combination of classical and custom strategies presented in this project provides valuable insights into the trade-offs in search algorithm design and lays a solid foundation for further enhancements in real-world routing applications.

8. References

[1] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2010.

[3] GeeksforGeeks, "Input Parsing and Graph Algorithms in Python," Available: https://www.geeksforgeeks.org/ (accessed: Month, Year).

[4] Real Python, "Understanding Data Structures in Python," Available: https://realpython.com/ (accessed: Month, Year).

[5] D. Poole and A. K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*, Cambridge, UK: Cambridge University Press, 2010.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[7] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.

[8] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.