Cover sheet for submission of work for assessment

UNIT DETAILS

Unit name	Introduction to Artificial Intelligent			Class day/time	
Unit code	COS30019	Assignment no.	2A	Due date	13/04/2025
Name of lecturer/teacher	Prof. Bao Vo				
Tutor/marker's name	Hy Nguyen				

Office use only

Faculty or school date stamp

STUDENT(S)

	Family Name(s)	Given Name(s)	Student ID Number(s)
(1)	Ly	Vien Minh Khoi	103844366
(2)	Hand	Gareth	104156787
(3)	Leong	Chun Wai	105239948
(4)	Bong	Christine Khai Ning	102787457

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) KHOI

(4) CHRISTINE

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/**

(2)	GARETH	(5)	
(3)	CHUN WAI	(6)	

Table of Contents

Instructions 4

Introduction 5

Features Implemented 6

Bugs Found 7

Missing Features 8

Testing 10

Insights 11

Research 13

- **Random Path Generator** 13
- **Graph GUI** 14

Conclusion 16

Acknowledgements/ Resources 17

Workload Matrix 19

Instructions

The assignment involves the development of 2 executable Python files: path_finding_algorithms.py and random_path_generator.py. Both of these files are included in the zip folder and can be run following these steps:

1. Extract the zip folder to a desired directory.
2. Navigate to the extracted folder and open the terminal by right clicking and choose Open in Terminal.
3. If there are no test case available, generate a test case either manually or by running the random_path_generator.py file: enter the command py random_path_generator.py (i) (-d), with:

a. (i) being the number of test cases required to be generated.
b. (-d) being whether the generated test cases should be drawn in a GUI, leave empty if no drawing required.

The command and the result of executing that command should look something like this:

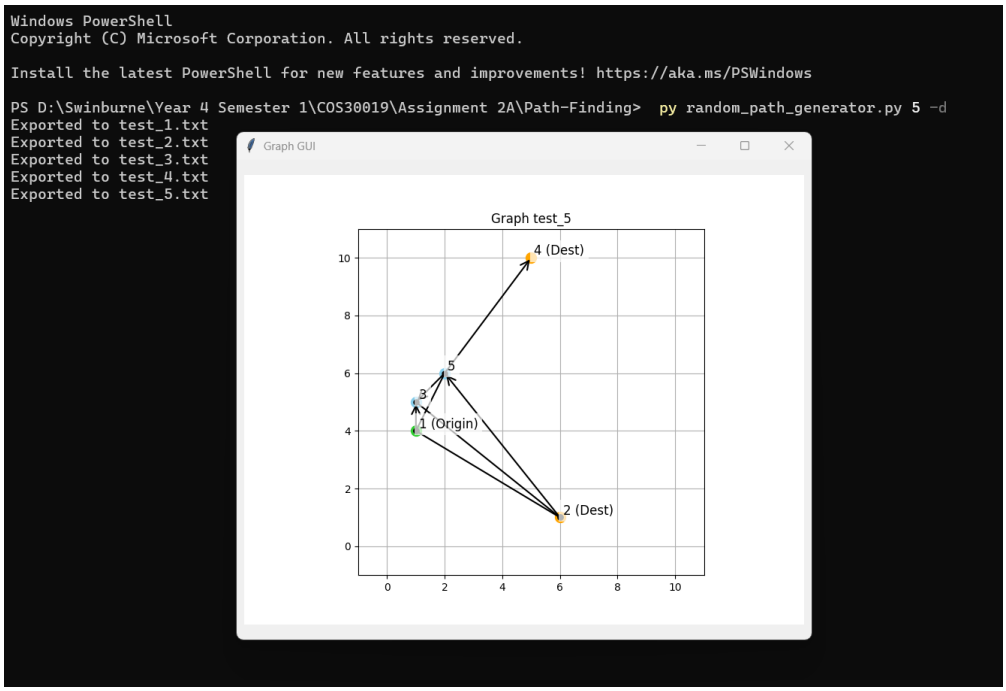

Figure 1. Result from running random_path_generator.py to generate 5 random graphs and draw the generated graphs.

4. Run the path_finding_algorithms.py file: py path_finding_algorithms.py (test_file_name.txt) (algorithm), with:

a. (test_file_name.txt) being the text file's name generated that is to be tested. Replace this with (-a) if it is required to test all text file in the current directory. The text file needs to be under a specific format for the command to work.

b. (algorithm) being the searching algorithm that is to be used to find the solution for the test file. This can be DFS, BFS, GBFS, AS, CUS1, CUS2, or (-a), with the last option being all implemented algorithms (in which case there will be 6 graphs generated).

The command and the result of executing that command should look something like this:

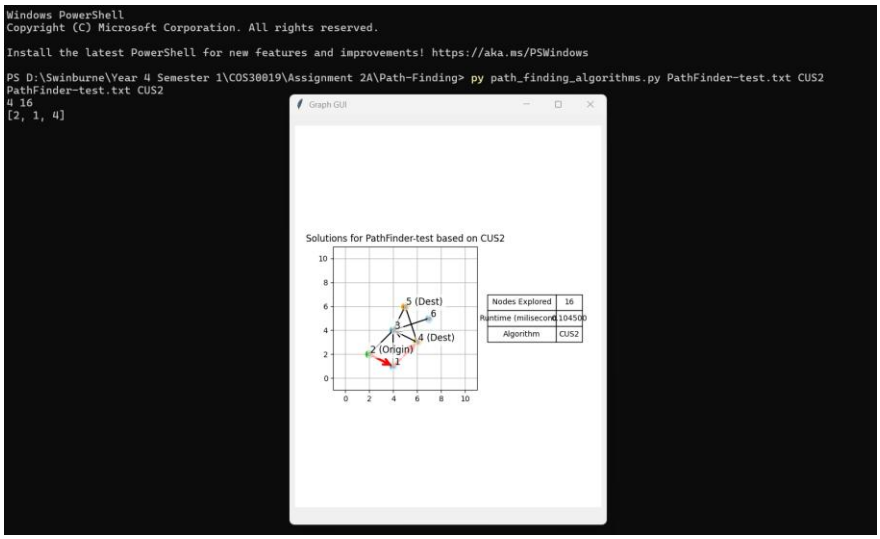

Figure 2. Result from running path_finding_algorithms.py using the algorithm CUS2 (IDA) on the PathFinder-test.txt test file.*

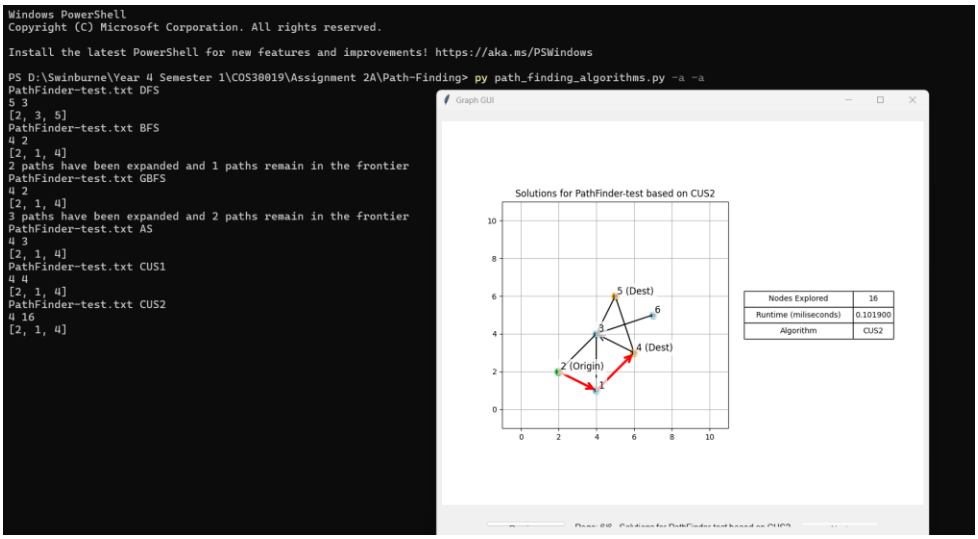

Figure 3. Result from running path_finding_algorithms.py using all implemented algorithm on all test files.

Introduction

Path finding is a common problem in the field of Artificial Intelligent (AI) where the best route needed to be found from an origin to one of the destinations on a map, which is formed as a graph containing nodes and edges. While nodes are points with specific Cartesian coordinate, edges connecting nodes that can

be traverse in one or both directions and have a specific cost assigned to it. The goal of this assignment is to use different searching algorithms to find the best path to one of the destinations using the information given. This information can just be what given by the problem initially, such as in the case of uninformed algorithms; or an estimate heuristic value to the goal nodes, like in the case of informed algorithms; both of which are explored and implemented in this assignment. These algorithms were listed and described in Table 1.

Table 1. Searching algorithms implemented.

Searching Algorithms	Description
Uninformed	
Depth-First Search (DFS)	Explores as far as possible before backtracking.
Breadth-First Search (BFS)	Explores all nearby nodes first before moving deeper.
Uniform Cost Search (CUS1)	Explores the path with the least cumulative cost.
Informed	
Greedy Best-First Search (GBFS)	Uses the estimated heuristic value to the goal to pick the most promising path.
A-Star (A*)	Combines the path cost and the heuristic value to the goal to find the best route.
Iterative Deepening A-Star (CUS2)	Preform DFS iteratively with the maximum AS heuristic function for each iteration increases.

These algorithms were tested on multiple graphs, which can be the sample provided (PathFinder-Test) or randomly generated using a custom program, to see which performs the best in terms of runtime, optimality, and number of nodes expanded. A Graphical User Interface (GUI) was also developed, demonstrating the graph visually as well as the path found and the performance metric for each algorithm. This report aims to explore the features implemented, the results of each algorithm, as well as the key insights observed from the result. It also focuses on the potential limitations in the implementation, along with some bugs still exist within the program.

Features Implemented

1. All Six Search Algorithms

We successfully were able to implement all six required search algorithms.

- DFS (Depth-First Search) for uninformed search.
- BFS (Breadth-First Search) for uninformed search.
- GBFS (Greedy-based-first Search) for heuristic based informed search
- A* (A-Star) for heuristic based informed search.
- CUS1 was Uniform Cost Search, which prioritizes the path with the lowest cumulative cost regarding how close the actual node is, utilizing a priority queue sorted by path cost.

- CUS2 was Iterative Deepening A-star Search, which is a memory efficient informed algorithm. It is a smarter version of the A-Star algorithm that uses less memory. Instead of keeping all the paths in the memory it slowly explores each path one at a time increasing the search limit until it reaches its goal.

2. Dynamic File Reading

The program reads graphs from a structured txt file. It then automatically parses all nodes, edges and the origin and destinations to the code. This allows us to have easy testing with multiple different maps by only changing the input text file.

3. Graph Visualization

We developed a fully functional GUI utilizing the tkinter and matplotlib to allow users to:

- Visually see the graphs layout including all the nodes, edges, origin, destinations and even the solution path.
- It highlights the origin in green, destination in orange and the solution path in red to allow for easy understanding of what the graph entails.
- A performance tab is right next to the graph which shows which algorithm has been used, runtime and how many nodes have been explored giving valuable insight into the respective algorithm.

4. Random Graph Generator

We created a test case generator to support automated testing.

- It creates random graphs with coordinates, edges, varied costs
- Randomizes the Origin and 2 destinations
- Exports all this into a correctly formatted txt file allowing it to be easily utilized in the main program when testing the search algorithms

Bugs Found

1. Minimal Error Handling

If anything like the txt file being inputted wrong or formatted wrong will cause the program to crash and throw multiple error codes. This doesn't cause any issues if everything is formatted and inputted correctly but could be handled better.

2. Fixed Graph

The graph is fixed using static grid which can provide issues if nodes are close together causing them to overlap and thus can be hard to distinguish between nodes. This can only cause a visual issue and wont effect anything on how the problem is solved through the algorithms.

3. Chance of Failing Graphs

There is a small chance that the graph generator creates a map where there is no possible link to origin to destination which causes a failure since there is no possible solution. This can be useful for testing a guaranteed failing test case however the graph doesn't display any clear way showing that there was no solution available, making it hard for the user to come to the conclusion that the graph didn't actually have a solution.

Missing Features

1. Exporting Feature

Currently, there is no way to export the results of each test case scenario to a useable format. Although you can take screenshots for each test case, an export feature could be useful for quick verification and comparison.

2. Automated Testing

The program provides the answers to each test case scenario, however there is no way other than manually checking to know if this provided solution is correct or if it's providing a wrong answer.

3. All Destinations

An idea was proposed of possibly making the program visit all destinations and find the shortest path. This is the research initiative that was originally required by the assignment. However, it was decided to not implement this feature due to substantial research gaps and efforts. In exchange, it was decided that the random graph generator and the GUI were to be implemented instead, both of which are further discussed in the Research section.

4. Test multiple but not all graphs

The current implementation only allows the graphs to be tested either one at a time or all at once. The feature that allows multiple but not all graphs to be tested against the searching algorithm can be useful to minimize runtime when only specific test cases needed to be verified.

5. Test multiple but not all algorithms

The current implementation only allows the graphs to be tested with either one search algorithm at a time or all of them at once. The feature that allows multiple but not all algorithms to be tested can be useful when only a few algorithms needed to be compared to the others.

6. Algorithms searching process visualization

Currently, the working principles of each algorithm can only be abstractly understood through their respective implementation or through their solutions and performance metrics when tested against a test case. It can be useful to graphically visualize each algorithm's searching process, either through animations or a series of graphs, for further clarification of their performance.

7. Encompassing GUI

There is a need for an encompassing GUI, where users can randomly generate test cases and perform their desired searching algorithms all in one location. This helps users to not have to constantly switching between commands to run the generator and the algorithm, which significantly reduces their time in verifying the test cases.

Testing

Table 2. Results of testing all path finding algorithms on the sample test case and 10 random test cases.

	DFS	BFS	GBFS	A*	CUS1 (UCS)	CUS2 (IDA*)
PathFinder-test	5 3 10 0.07ms [2, 3, 5]	4 2 10 0.06ms [2, 1, 4]	4 2 10 0.28ms [2, 1, 4]	4 3 10 0.36ms [2, 1, 4]	4 4 10 0.07ms [2, 1, 4]	4 16 10 0.16ms [2, 1, 4]
test_1	5 1 14 0.03ms [4, 5]	5 1 14 0.04ms [4, 5]	5 1 14 0.26ms [4, 5]	5 1 14 0.31ms [4, 5]	5 3 14 0.05ms [4, 5]	5 8 14 0.08ms [4, 5]
test_2	2 4 8 0.04ms [5, 2]	2 1 8 0.03ms [5, 2]	2 1 8 0.41ms [5, 2]	2 1 8 0.29ms [5, 2]	2 3 8 0.05ms [5, 2]	2 7 8 0.07ms [5, 2]
test_3	4 1 8 0.02ms [3, 4]	4 1 8 0.03ms [3, 4]	4 1 8 0.32ms [3, 4]	4 1 8 0.37ms [3, 4]	4 3 8 0.04ms [3, 4]	4 6 8 0.07ms [3, 4]
test_4	NS 1 0.01ms None	NS 1 0.03ms None	NS 1 0.01ms None	NS 1 0.02ms None	NS 1 0.03ms None	NS 1 0.02ms None
test_5	2 3 18 0.05ms [1, 3, 2]	2 2 32 0.06ms [1, 5, 2]	2 2 18 0.28ms [1, 3, 2]	2 3 18 0.28ms [1, 3, 2]	2 5 18 0.06ms [1, 3, 2]	2 32 18 0.22ms [1, 3, 2]
test_6	3 1 10 0.02ms [4, 3]	2 1 13 0.03ms [4, 2]	2 1 13 0.27ms [4, 2]	3 2 10 0.29ms [4, 3]	3 3 10 0.06ms [4, 3]	3 16 10 0.17ms [4, 3]
test_7	4 2 15 0.04ms [5, 1, 4]	4 2 15 0.04ms [5, 1, 4]	4 2 15 0.31ms [5, 1, 4]	4 2 15 0.21ms [5, 1, 4]	4 3 15 0.06ms [5, 1, 4]	4 9 15 0.11ms [5, 1, 4]
test_8	1 0 None 0.01ms None	1 0 None 0.03ms None	1 0 None 0.18ms None	1 0 None 0.2ms None	1 0 None 0.01ms None	1 0 None 0.01ms None
test_9	3 1 5 0.03ms [5, 3]	1 1 6 0.05ms [5, 1]	1 1 6 0.29ms [5, 1]	3 1 5 0.34ms [5, 3]	3 2 5 0.05ms [5, 3]	3 6 5 0.09ms [5, 3]
test_10	NS 3 0.04ms None	NS 3 0.04ms None	NS 3 0.05ms None	NS 3 0.06ms None	NS 3 0.04ms None	NS 1963169 17779.35ms None
Average	**1.78** **9.78** **0.03ms**	**1.22** **11.78** **0.04ms**	**1.22** **10.22** **0.29ms**	**1.56** **9.78** **0.29ms**	**2.89** **9.78** **0.05ms**	**11.11** **9.78** **0.1ms**

Table 2 shows the test results of the sample test case and 10 random test cases for all 6 algorithms. The bottom row of the table shows the average number of nodes explored, path cost and runtime of each algorithm excluding the test cases where it has no solution. The results are displayed in the format described in Table 3.

Table 3. Result display format.

Row Number	Result for each test	Average result
First row	(destination) (number of nodes explored) (path cost)	(number of nodes explored)
Second row	(runtime)	(path cost)
Third row	(path)	(runtime)

Within the random test cases generated, there are some special cases to be considered:

- No edges connected from the origin: In test case 4, the origin node has no outgoing edges to expand which means it is isolated. All algorithms provided the same result of "No solution" correctly. The program was terminated immediately because there were no neighbors found.
- No edges connected to the destination: In test case 10, the destination node has no incoming edges which means the other nodes are unable to expand to it. All algorithms provided the same result correctly, "No solution". However, the algorithm IDA* has a large number of nodes explored which will be explained in Insights. The rest of the algorithm has the same number of nodes explored because there are only 3 nodes that are reachable.
- Origin is the destination: In test case 8, the origin node is same as the destination node. All algorithms provided the same output because all of it checks whether the starting node (origin) is the goal node (destination) before it is entering the iteration to expand the neighbors.

Insights

DFS expands the nodes vertically, meaning that the algorithm expands the entire depth of a branch in the search graph first before moving on to other branches. DFS did not have any outstanding result as seen in Table 2. Even though the algorithm has the lowest path cost, this was concluded to be just a coincidence due to DFS does not taking in path cost as a consideration in its expansion decision. This makes DFS suitable for deep and narrow graph, which most of our test cases are not.

On the other hand, BFS expand its search horizontally, which means it expands all the neighbor branch in a level before continuing deeper in any branch. Based on the results shown in Table 2, it can be seen that BFS has the best average number of nodes explored and runtime despite having the worst path cost. This is because deploying the algorithm guarantees to find the shortest path without taking in path cost as a consideration. Therefore, BFS is more suitable in small and unweight graphs, those of which require the shortest path to the destination.

Conversely, GBFS is an informed searching algorithm that only uses the estimated heuristic function to expand the path, which allows it to know how close it is to the destination. In Table 2, this algorithm has the same lowest number of nodes explored as BFS when tested against our test cases. Despite this, it is

more inconsistent compared to BFS because it may be misled by a bad heuristic function, which guide the search to a path that is longer despite having lower nodes with lower heuristic value.

Another informed searching algorithm is A*, which combines the cumulative path cost of the node from the origin with its estimated heuristic value to the goal, allowing it to have the most efficient path cost with the lowest number of nodes explored. A disadvantage of this algorithm is its runtime, as A* will need to compare the values of both path cost and heuristic function between different paths to be taken. As a result, the algorithm has the worst runtime in Table 2. A* is more suitable in large and weighted graphs because it uses a method of estimation that considers all possible parameters for a node. This helps to avoid unnecessary exploration in the graph and thus provides the most efficient and optimal path.

One of the custom searching algorithms chosen to be implemented is UCS, which is an uninformed algorithm that utilizes the cumulative path cost from the origin to make decisions. Due to lack of an informed heuristics function, UCS has no direction guidance and must explore more node to compare the cumulative path cost. This result in UCS being one of the algorithms with the lowest path cost but also having a relatively high number of nodes explored compared to others.

The last implemented custom search algorithm is IDA*, an algorithm that combines the heuristics nature of A* with the low space complexity of limited-depth DFS. This algorithm iteratively explored the depth of each branch of a search tree with the largest heuristic value in each depth level as the threshold. Due to its recursive nature, meaning that the algorithm will start the search from the origin again with an increase threshold whenever the threshold is met, the number of nodes explored of this algorithm grows exponentially with each iteration. This is reflected in this performance metric being so significantly higher than other algorithms in Table 2, despite its runtime resides on the lower side. it will repeatedly explore the entire graph until it maxed out the possible path cost in the graph. Furthermore, IDA* also struggles in cases where no edges are connected to the destination nodes. Because the algorithm relies on the threshold value being reached as a condition to terminate the search, it continuously iterates through a branch indefinitely from the origin without ever reaching the destination. This results in a massive runtime and number of nodes explored in the last test case of Table 2. Ultimately, IDA* is most suitable for finding the most optimal path with very little memory consumption (low space complexity), but in exchange it required a significant number of nodes to be explored before reaching the destination, resulting in potentially large time complexity in some situations.

In conclusion, A* is better among these algorithms because it has the most balanced and practical results. In real-world scenario, people tend to look for algorithm that can achieve multiple purposes. A* not only can be used to find the optimal path but also reduce the unnecessary expansion. In the test results, A* consistently provides the least number of nodes explored among all the algorithms that provided the most optimal path despite having a little longer runtime.

Research

As described in the Features Implemented sections, the assignment also includes a random path generator program as well as a GUI that shows the visual representation of the generated graphs and the solution path. These extended features that are not parts of the compulsory requirements are considered the team's research effort instead of a solution path covered all destinations as explained in the Missing Features section.

Random Path Generator

The random path generator program contains 3 methods, each of which having very distinctive functions. The first method, denoting random_graph, is responsible for randomly generating graphs as a concatenated string that can be read by the searching algorithms to create a full graph object. This method firstly created 5 nodes whose name range from 1 to 5 and coordinate are tuples of random integers ranging from 0 to 10. These nodes are stored as items in the "nodes" dictionary, which then gets iterated over to generate random edges connecting them to one another. For each node in the "nodes" dictionary, a random integer between 0 and 3 is assigned to specify how many outgoing connections the node can accept. During the iteration, the current node is then randomly connected to other nodes up to the allowed number of connections, resulting in directed edges from the current node to randomly selected target nodes. Subsequently, a cost is calculated between connecting nodes based on the formular $ceilling(\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2})$, which is derived from the sample graph's path cost being roughly twice the distance between the nodes. This process can also randomly happen in reverse, which is when the node in the current iteration receives incoming connections from these nodes. This is controlled by a Boolean flag that is randomly assigned either "true" or "false". The cost of these inverse connection can be the original cost or the original cost plus 1. The edges created in this way are stored in the "edges" dictionary under the format defined in the assignment requirements, which is a tuple of the connected nodes (from the first node to the second node) as the key and the cost as the value.

Similarly, the origin and destinations of these graphs are also randomly selected from the nodes created. The number of destinations available is randomized between 1 and 2, with the latter can be changed to any numbers below the number of nodes available minus 1 (which is the origin). A destination can be selected randomly from the list of keys of the "nodes" dictionary if the chosen node is not the origin and has already been added to the destinations list. This process can happen up to the number of destinations randomized earlier, before the destinations chosen are stored as integer in a list data structure. The dictionaries, integer, and list created are subsequently parse and converted to a single string that have the specific format required, which gets returned along with all the data structure created.

The second method, denoting export_graph, is responsible for exporting the graph string generated to text file, whose name is dynamically updated based on the name of the text file already available in the current

directory. This and the first method are directly used in the third method of runGenerator, which parse the CLI argument to determine the number of graphs are to be generated. Furthermore, this method also draws the generated graph using the GUI class developed based on user's command. Specifically, if the second CLI argument is "-d", the method will extract the parameters needed from generated graphs, which are stored in a "maps" dictionary with the graph's id being the key and its content (nodes, edges, origin, and destinations) being the value. These parameters can then be passed into the class's constructor to generate a root, whose GUI loop was called, prompting the GUI to initiate. This shows all the graphs generated in a contained GUI for visual clarification. The development of the graph GUI is further explained in the next subsection.

Graph GUI

The graph GUI is a class utilises Python's tkinter and matplotlib libraries to initiate a GUI containing all the graphs and solutions generated. The GUI composed of 2 frames: the navigation frame and the figure frame, with the latter further divided into the graph tab and the performance metrics tab. The dimensions of these frames as well as of the GUI's window can be dynamically changed in the constructor, improving the flexibility and scalability of the application. The class encapsulates all functions necessary to generate and control the GUI. This ensures that the same logic can be used across multiple programs without the needs for re-implementation, which can be seen in the fact that the class was used in both random_path_generator.py and path_finding_algorithms.py.

One of the encapsulated functions is draw_graph, which takes in the data structures relevant to a graph (nodes, edges, origin, destinations, and title) as parameters, looping through each of them to determine the required components, and plot them on a figure with annotated special vertices (like origin and destinations) and edges (like directed or undirected edges). Similar logic can be seen in the draw_solution function with two additional parameters of the solution path and metrics. While the former gets parsed and drawn as red arrows from the origin to one of the destinations on the GUI; the latter describe the performance metrics of the method chosen, including variables such as runtime, number of nodes explored, and algorithm name, all saved in a dictionary. Here, the metrics parameter is parse and drawn in a table with default values initiated, which is situated in the performance metric tab of the figure frame. The general layouts of the GUI for both drawing generated graph and solution can be seen in Figure 4 and Figure 5.

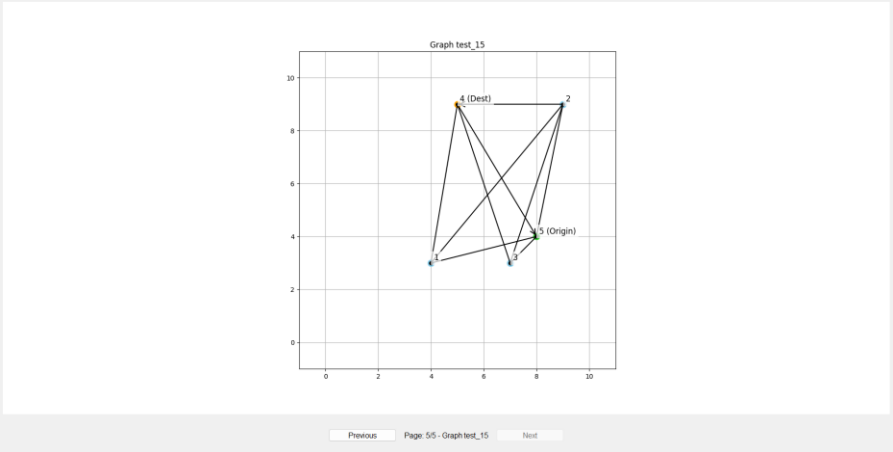

Figure 4. The GUI layout when using the draw graph function.

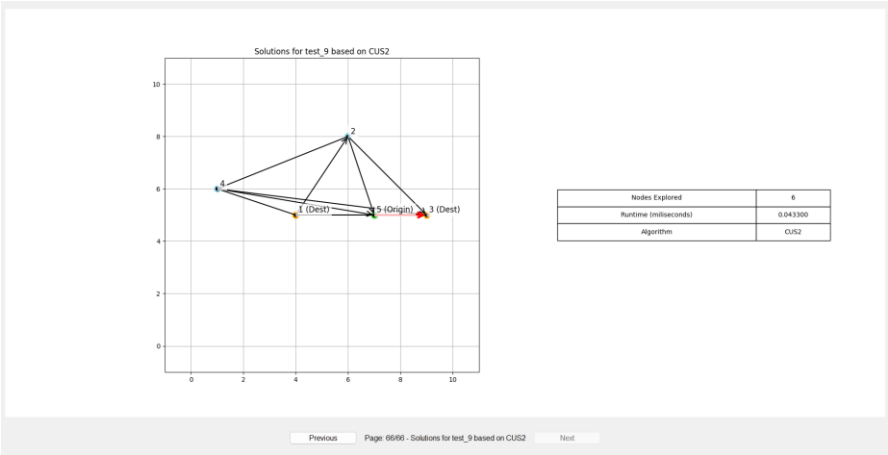

Figure 5. The GUI layout when using the draw solution function.

Both of these functions return the figure object, which is displayed in the graph tab of the figure frame through the function display_current_graph. For multiple graphs, the function displayed them in multiple pages, which can be navigated via the buttons in the navigation frame situated right bellow the figure frame. The logic to control these buttons can be found in next_page, prev_page, and update_button functions. Moreover, in order for the program to terminate when closing the GUI window, another function called on_closing was added, which explicitly tells the system to exit when called. The class needed to be constructed and used in conjunction with the tkinter root, which can be used to start the main GUI loop after created. The root's protocol can be defined before this process, which used the class's on_closing function as the parameter when the event "delete window" is executed.

Conclusion

This assignment involved implementing and testing six search algorithms: DFS, BFS, GBFS, A*, UCS, and IDA*. Through testing on various graphs, we found that each algorithm has distinct advantages and limitations. DFS and BFS were efficient but didn't always find optimal paths. GBFS performed well but could be misled by poor heuristics. A* stood out as the most balanced option, consistently finding optimal paths while keeping node exploration reasonable. UCS achieved optimal paths but explored more nodes, while IDA* had excessive node exploration due to its iterative nature.

Our implementation successfully handled edge cases and included helpful features like graph visualization and random test generation. Overall, this project demonstrated that while A* was generally most effective, the best algorithm choice depends on specific problem requirements and constraints.

Acknowledgements/ Resources

COS30019 Lectures and Tutorials

The weekly lectures and lab sessions were crucial in helping us understand tree-based search strategies, node expansion, and heuristic-based reasoning. These concepts formed the foundation of our implementation, especially for structuring the Node and Problem classes and integrating goal tests, cost functions, and heuristics.

GeeksForGeeks – Algorithm References

Depth-First Search (DFS)

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

This article helped reinforce the recursive and stack-based nature of DFS. It was especially useful for handling deep graph traversal and understanding how to prevent infinite loops using a visited set.

Breadth-First Search (BFS)

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

This page helped guide the queue-based logic for implementing BFS. We used collections.deque to efficiently implement the FIFO frontier, and we mirrored the node expansion style shown in the example. The logic also helped us ensure that nodes are explored in order of depth (i.e., all immediate neighbors before deeper ones).

Greedy Best-First Search (GBFS)

https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/

https://www.geeksforgeeks.org/greedy-best-first-search-in-ai/?ref=ml_lbp

These two GBFS guides shaped our understanding of how to build a heuristic-driven search. We adapted the example to implement GBFS using best_firstgraph_search(), where f(n) = h(n). The use of heapq and a memorized heuristic help us efficiently guide the search towards its cost.

A* Search (AS)

https://www.geeksforgeeks.org/a-search-algorithm/

This was critical in designing our AS method. The explanation of combining g(n) and h(n) to compute f(n) helped us structure the lambda function for prioritizing nodes and optimizing for both cost and estimated distance.

Uniform Cost Search (UCS)

https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

This example guided our UCS implementation using heap. We followed the structure of comparing path costs and replacing more expensive paths in the frontier when needed.

Iterative Deepening A (IDA)

https://www.algorithms-and-technologies.com/iterative_deepening_a_star/python

We used this resource to gain a clearer understanding of how IDA* works and to help shape our CUS2 implementation. It follows the same pattern of threshold-based recursion using f(n) = g(n) + h(n), which aligned closely with the structure we applied in our own code.

Workload Matrix

	Ly Vien Minh Khoi (103844366)	Gareth Hand (104156787)	Christine Khai Ning Bong (102787457)	Chun Wai Leong (105239948)
Instructions	✓			
Introduction	✓		✓	
Features/Bugs/Missing	✓	✓		
Testing	✓			✓
Insights	✓			✓
Research	✓	✓		
Conclusion			✓	