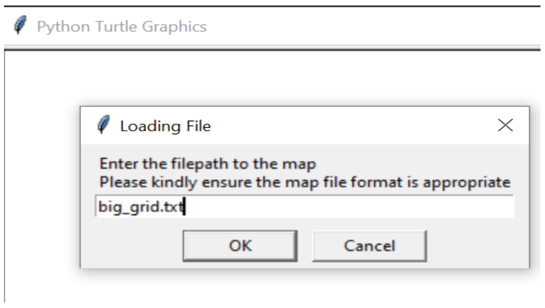
DATA STRUCTURES & ALGORITHMS CA2 Report

By **GROUP 2 DAAA2B02**

Team Members:
Lim Hur (2112589)
Yee Hang (2112675)

1 Instructions to operate program

Please kindly use the command “python main.py” to run the program at the project directory. Next, a prompt would appear asking for the file path to the map. Please note that the map file should be in the same directory. Relevant validations are done for file input.

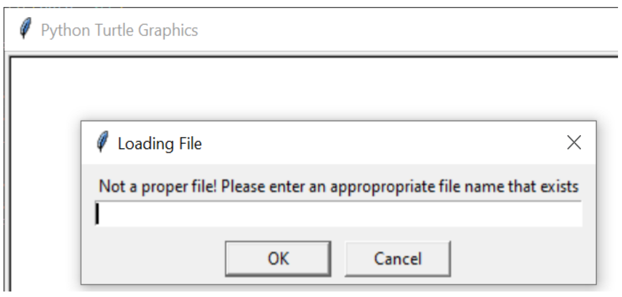

Python Turtle Graphics

Loading File

Enter the filepath to the map
Please kindly ensure the map file format is appropriate

big_grid.txt

OK Cancel

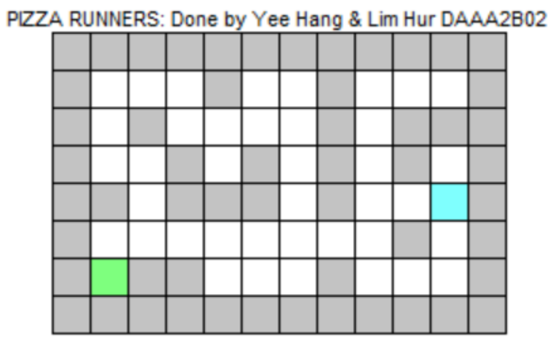

Python Turtle Graphics

Loading File

Not a proper file! Please enter an appropropriate file name that exists

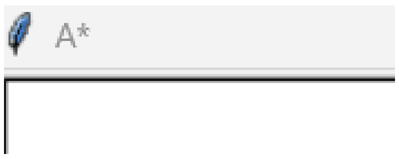
OK Cancel

After loading any map of your choice, we should expect such a page, with the team members name displayed on the top. :

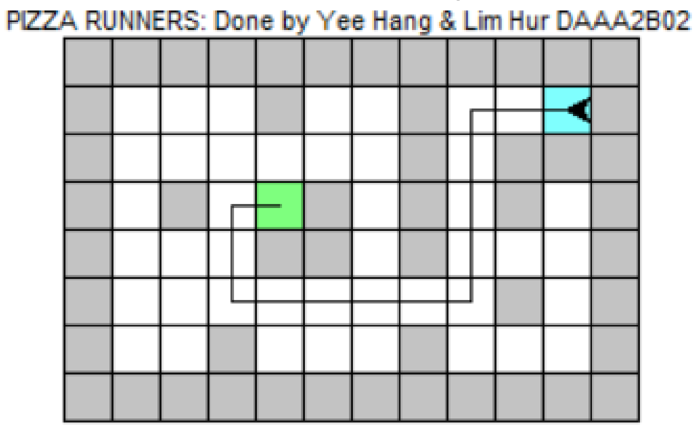

PIZZA RUNNERS: Done by Yee Hang & Lim Hur DAAA2B02

Press Tab key to change the algorithm (Default algorithm is Left Hand Rule, second algorithm is A Star)

LeftHand


A*

Press 'f' for the algorithm to initiate path finding and display the drone and the no. of actions taken on the screen. Example:

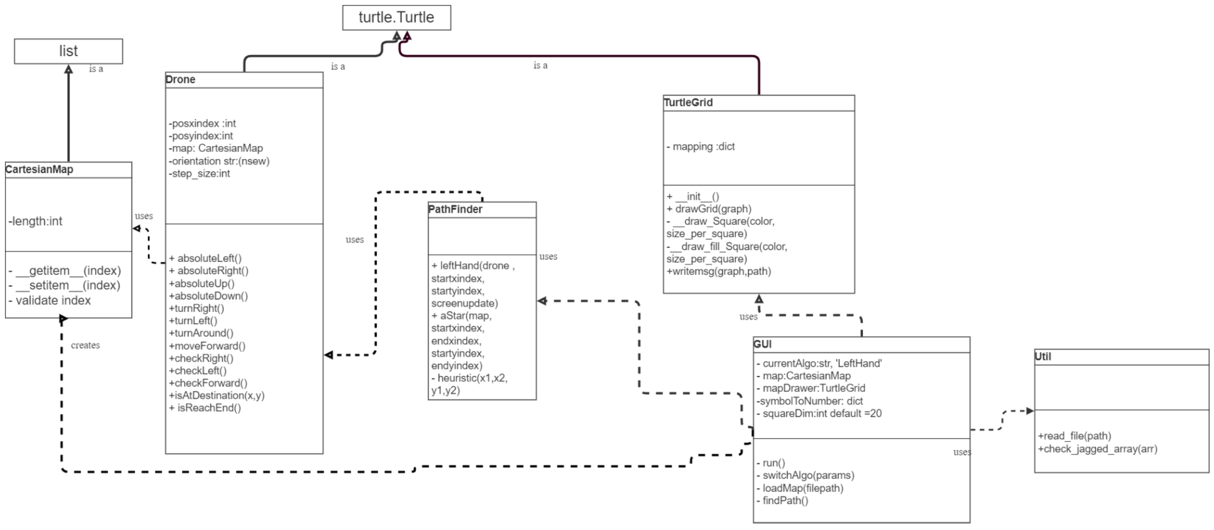

PIZZA RUNNERS: Done by Yee Hang & Lim Hur DAAA2B02

Actions taken by A*: 14

KEY COMMANDS IN APPLICATION

Tab key	Switch between Left hand rule and A*
n key	Load a new map (only available after the algorithm's stop condition is met)
f key	Use algorithm (A* or LHR) to find path
q key	Quit the program

2 Class Design & Object Oriented Programming



OOP Principles	Description and Discussion
Inheritance	*Cartesian grid* ***is*** *a list* *TurtleGrid* ***is*** *a turtle.Turtle* *Drone* ***is*** *a turtle.Turtle* TurtleGrid and Drone **subclass the python's turtle.Turtle** to inherit functionality to the turtle class to draw the maze and the drone on screen. Cartesian Grid data structure **subclass the python's list, for custom indexing explained later**
Encapsulation	Helper methods within a class is **private** Methods that are to be used by other classes are **exposed** All properties of the class are set to **private. However,** properties that need to be used by

	another class are exposed via **getters**. For example, the current position of the drone is exposed via **getters** but no setter is needed for them , as no other classes need to use the setter, thus making the current position of the drone read only properties

Class	General Description	Methods
GUI	Handles writing and drawing to the screen by using turtle instances Also acts as a controller , as it uses a turtle screen instance to detect and respond to user input	**run()** To start the whole program , Starts a turtle screen , setup event handling and starts the mainloop
		switchAlgo() Changes the path finding algorithm between A* and Left Hand Rule . Called every time the “Tab” key is pressed
		Load_map() - Called every time ‘n’ key is pressed - Ask the user for relative file path to map - Validates the user input (ensure file in correct format / file exists) - Builds and draws the files on screen
		find_path() - Uses the current map and current pathfinding algorithm loaded in to find the path and print path to screen

Turtle Grid (inheritance)	Turtle Grid subclasses the turtle. (TurtleGrid is a Turtle) Turtle() class and adds the functionality of drawing the loaded map on screen	**Draw_grid(map)**
		- Draws the map grid on screen
		
		drawSquare(color, size_per_square)
		- Helper method that draws unfilled square, for drawing open space that the drone can walk through
		
		drawFilledSquare(color,size_per_square)
		- Helper method that draws filled square based on <size_per_square>, used for drawing building, and the start and end position

Cartesian Grid	Subclass the python list. Cartesian Map is a list() representing the 2d array map. It allows for cartesian indexing which is more inline with the turtle grid coordinate system, making indexing the map easier.	**OPERATOR OVERLOADING** **__getitem__(index):** Allows for indexing with cartesian coordinates maze[x,y] **__setitem__(index):** Map[x,y] = value Sets the value of the map[x,y] to value Overload these 2 methods the default list indexing to accept 2 values in the map[x,y] We **overload the __getitem__ and __setitem__ operators** to allow indexing with more than one index and allow indexing using cartesian grid(one index for x another for y)
Drone	Subclasses the turtle.Turtle (Drone is a Turtle) This class is a drone that navigates given a map. It keeps track of its current location in the map and handles the indexing of the map. The pathfinding algorithms use the high level methods such as checkLeft() and turnLeft() and the algorithm does not have to handle the current direction the drone is facing (north ,south,east , west).	**__absolute (left() / right() /up() / down())** Helper methods that return the cell on the map directly the left, right up and down of the drone's current location in the map **turn(left(), right() , around())** Uses the .left(90) or .right(90) of the parent class (turtle.Turtle) to turn left and right, and updates the current orientation the drone is facing (North , South , East or West) **moveForward()** Move one step forward in the grid **check(Right(), Left(), Forward())** Returns a boolean of whether the cell on the right left forward is a building This takes into the orientation the drone is currently facing, If .checkRight() is called while the drone facing east , the cell to south is returned If .checkRight() is called while drone facing west , the cell to north is returned **isAtDestination()** Checks if current position of drone is the same position as the end position

Pathfinder	Adds support for path finding methods such as A Star and Left Hand rule	**LeftHandRule()/Astar() (static method)**
		- Implements the respective algorithms, using the map, starting position, ending position
		heuristic() (static method)
		- Implements manhattan distance as heuristic for astar as moving in diagonal cells is not allowed in this application

3 Summary of Data Structures used

In this section, we will share with you our reasoning on the choice of data structures

Before finalising our data structures, we had to first evaluate the pros and cons of our data structures. We developed and compare the use of different data structures in the different scenarios scenarios:

DATA STRUCTURES USED: PRIORITY QUEUE, CARTESIAN GRID, DICTIONARY

Data Structure	**Usage and thought process**
Cartesian Grid **Scenario:** Support for drone movements and Grid representation	**Movement of drone and grid representation:** To provide an **intuitive and efficient** way to quickly search the cell type (building, open road or not), we developed a **cartesian grid** to represent the maze with support for cartesian coordinates indexing (x, y coordinates). Due to the fact that setting and indexing of cells in a cartesian grid is **O(1)** time, constant time, our Drone is able to **swiftly** search the grid space, and avoid paths that contain a building. However there is another data structure that we considered: nested 2d python list; However, the shortcomings are the following: the rows on the bottom of maze have a lower y index than the rows on top , which is not **consistent with the turtle coordinate system**. This might cause indexing errors in the future, and make the program harder to maintain.

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)

(0,2)	(1,2)	(2,2)

X and y indexes using a default list, (0,0) is at the top left hand corner which is not consistent with the position for origin in cartesian coordinate system

(0,2)	(1,2)	(2,2)
(0,1)	(1,1)	(2,1)
(0,0)	(1,0)	(2,0)

X and y indexes using a Cartesian Grid, (0,0) at the bottom left corner which is more inline with the cartesian ***coordinate system of turtle***

An alternative way to store the map is to use a list to keep track of **all the coordinates with buildings**. In algorithms (a star or left hand rule), a membership check using python's **in** operator is used. Checking whether the cell in the list (checking whether the cell is road) has O(n) time complexity. This will slow down the algorithms, when the **map is big**, more time is required to check membership of **each cell as there are a lot of buildings.**

Each cell of the Cartesian Grid contains a 0 (representing buildings) or 1 (walkable terrain). By using an equality check with indexing (if maze[x,y] == 0:) , it has O(1) time complexity to check if a cell is road.
Hence, we went with the concept of cartesian grid to facilitate efficiency.

Dictionary **Scenario:** To update the current direction/ orientation that the drone is facing while it is moving through the map.	The alternative to using a **dictionary** to keep track of orientation is using **nested if else** statements to keep track of the current orientation of the drone leading to **poorer code readability and maintainability.** Setting and getting items from the dictionary has O(1) time complexity, in constant time, showing that **performance is not sacrificed**.
Priority Queue	**Priority queue** is used for our A star algorithm. It is a type of queue that arranges elements based on their priority. Items with higher priority values are retrieved before items with lower priority values.

Scenario: Keep track of the possible nodes that the A* Algorithm considered.	In the main program loop of A star, neighbouring nodes of the current node are kept tracked of, and in the start of the loop, the node with the lowest heuristic (closest to the destination) becomes the current node. The priority queue is suitable as it uses a heap in the back end. Retrieves the item with lowest f score in **O(log(n))** time (to satisfy heap property after pushing) Pushing item with lowest f score in O(log(n)) time Another data structure we thought of is using a **list**; However, the list had shortcomings: getting the node with the lowest f score is done has **O(n)** time complexity which is a lot slower that priority queue of **O(log(n))**. As priority queue is **more time efficient** in getting the node with lowest f score, it is used.

4 ALGORITHMS

4.1 Left Hand Rule

Intuitive PseudoCode to illustrate the main idea of LHR algorithm:

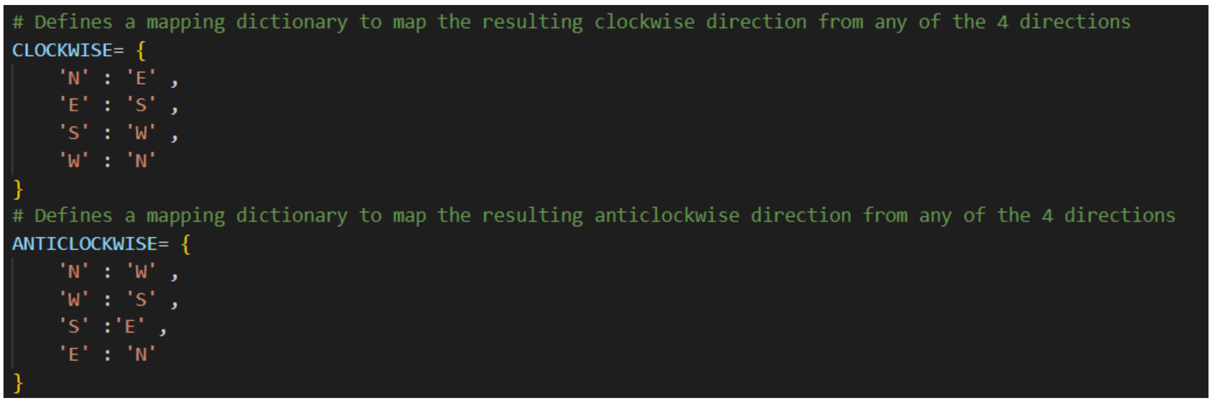
```
If left side is not building:
    Turn left
    When building is not reached:
        goForward
    Set starting position to the current position
While True :
    If destination reach:
        Return steps_taken
    If no solution
        Return 'No path found'
    If leftside is not building:
        Turn left and go forward
    Else if the front is not building:
        Go forward
    Else if the right side is not building:
        Turn right and move forward
    Else:
        Turn around and move forward
```

Stop condition for LHR:

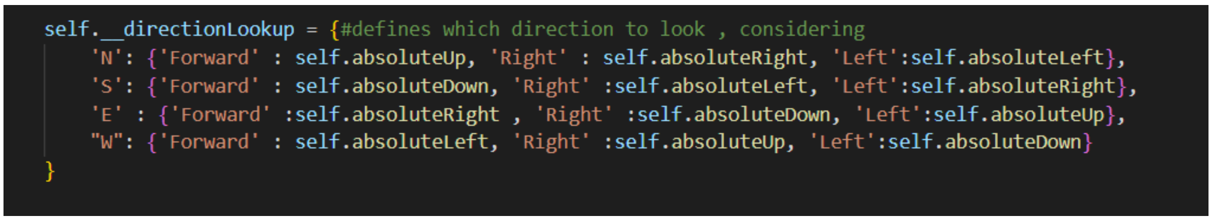
Left hand rule stops when it reaches a destination (path found) OR when it takes the same path twice, and returns to the same starting position as defined above (No path found)

Implementation of Left Hand Algorithm

As the algorithm requires the concept of turning left and right, it is important to keep track of which orientation (North, South, East or West) the drone is facing, as that makes it possible to index to the correct cell.
In the drone class, these class and instance properties are used to update the current orientation with the new orientation .



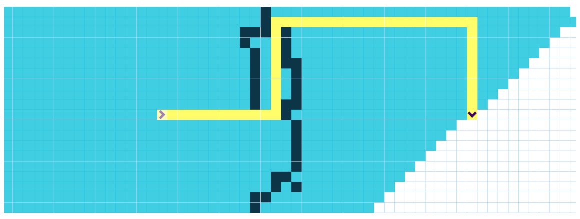
```
# Defines a mapping dictionary to map the resulting clockwise direction from any of the 4 directions
CLOCKWISE= {
    'N' : 'E' ,
    'E' : 'S' ,
    'S' : 'W' ,
    'W' : 'N'
}
# Defines a mapping dictionary to map the resulting anticlockwise direction from any of the 4 directions
ANTICLOCKWISE= {
    'N' : 'W' ,
    'W' : 'S' ,
    'S' :'E' ,
    'E' : 'N'
}
```



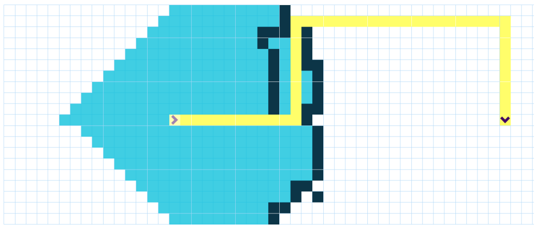
```
   self.__directionLookup = {#defines which direction to look , considering
       'N': {'Forward' : self.absoluteUp, 'Right' : self.absoluteRight, 'Left':self.absoluteLeft},
       'S': {'Forward' : self.absoluteDown, 'Right' :self.absoluteLeft, 'Left':self.absoluteRight},
       'E' : {'Forward' :self.absoluteRight , 'Right' :self.absoluteDown, 'Left':self.absoluteUp},
       "W": {'Forward' : self.absoluteLeft, 'Right' :self.absoluteUp, 'Left':self.absoluteDown}
   }
```

This allows for better code readability (the alternative way to keep track of direction is to use very nested if else statements) without sacrificing the efficiency as the indexing of the python dictionary is generally a O(1) time operation and hash collisions are not very likely to occur as these have a small number of elements and they are static , and no extra key value pairs will be added to them .

4.2 A* Pathfinding



Dijkstra pathfinding



A* pathfinding

Above image compares the djikstra pathfinding vs a* pathfinding. We see that A* explored lesser nodes given the smaller search space (in blue), but still reaching the destination in the shortest path. Djikstra on the other hand, explored more nodes, while still reaching the shortest path.

Comparison (theory)

For the second algorithm, we have **compared the Djikstra algorithm and a* algorithm for finding the shortest path**. The difference between the 2 is that A* is an extension of Djikstra and it tries to look for a more efficient path using a heuristic function, which is an **estimation of cost to reach the destination from any given**

node. A* uses the heuristic to prioritise searching the nodes that are closer to the destination, thus decreasing the search space (decreasing number of nodes to search before reaching a solution), as shown in the diagram above. On the other hand, **Dijkstra's explore all possible paths**. One thing to note is that the heuristic function has to be admissible for A*. Given the difference in the efficiency in exploration, and the existence of a heuristic function for a more informed search for A*, A* is our **choice of second algorithm.**

A* is **complete** such that (it finds a path if one exists) and **optimal** (always finds the shortest path) if we use an admissible heuristic function. (explained below)
A* has two cost function
g(n): the cost to reach node n
h(n): approximate cost from node n to goal node (heuristic function). This heuristic function should never overestimate the cost, meaning the real cost to reach goal node from node n should be greater than or equal h(n).

For this assignment, we will be using the manhattan distance as our admissible heuristics as we are moving to adjacent cells only and not diagonal cells.

Optimization for A* algorithm

Another optimization is we used a 2d Cartesian grid of booleans ‘visited’ to keep track of whether the element is in the ‘opened’ priority queue. Instead of checking for membership using the **in** keyword (which is **O(n) operation**) , we index the grid in O(1) time and check the value of the boolean.
This uses more memory but increases path finding speed as checking for membership of the opened is a common operation.

Summary of challenges faced

The main group work related challenge we encountered was syncing the most updated project files with one another. So when one of us changes the code, we would have to then send the entire zipped folder to our partner for them to add their portion. This was a huge hassle given that if 2 of us edits the source codes at the same time, then combining our source codes would be a huge hassle as we may have to scrutinise line by line for some files so as to incorporate both our changes in the final version. However, this problem was alleviated then by using Github and its branching feature, for collaboration. As a result, we were able to collaborate more efficiently.
In terms of technical problems, both of us had to face quite a steep learning curve in learning the necessary libraries such as turtle, as we were not very well versed with turtle initially apart from the practicals. However, after further research and guiding videos, it became more intuitive to us, allowing us to grasp the concepts of the turtle graphics API.

Key takeaways & learning achievements from CA2

As a group, our key takeaways are the following:

- This assignment provided us with greater clarity and appreciation to how efficient path finding is critical in real world scenarios/problems.
- We are also able to better relate on how algorithms such as A*/ left hand can be applied to existing problems in our day to day lives.
- For technical aspects, we also learnt the importance of developing efficient and relevant data structures, that would ultimately affect the time complexity of our solution. To do this, we needed to think logically and consider our problem at hand in different perspectives.
- Overall, we feel that the assignment (including individual features) tasked has broadened our horizons into the exciting study of data structures and algorithms and its great importance in the study of computer science.

Contributions:

Yee Hang	**Lim Hur**
- A* Algorithm - Drone (A star) - GUI Prepare map, find path - Class Design , discuss with partner - Discuss and do turtle grid with partner - Comments and encapsulation for individual components - Report Writing	- Left hand algorithm - Discuss to do a* with partner - Cartesian Grid - Class Design , discuss with partner - Drone (left hand) - GUI.load map method - Comments and encapsulation for individual components - Report Writing

APPENDIX

REFERENCES

https://en.wikipedia.org/wiki/A*_search_algorithm

https://stackoverflow.com/questions/16246026/is-a-star-guaranteed-to-give-the-shortest-path-in-a-2d-grid

https://clementmihailescu.github.io/Pathfinding-Visualizer/#

https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

https://stackoverflow.com/questions/13031462/difference-and-advantages-between-dijkstra-a-star

https://en.wikipedia.org/wiki/Admissible_heuristic

https://github.com/tonypdavis/Python-maze-solving-program-LHR/blob/master/LHR%20alogrithm%20with%20images.pptx (ppt slides)