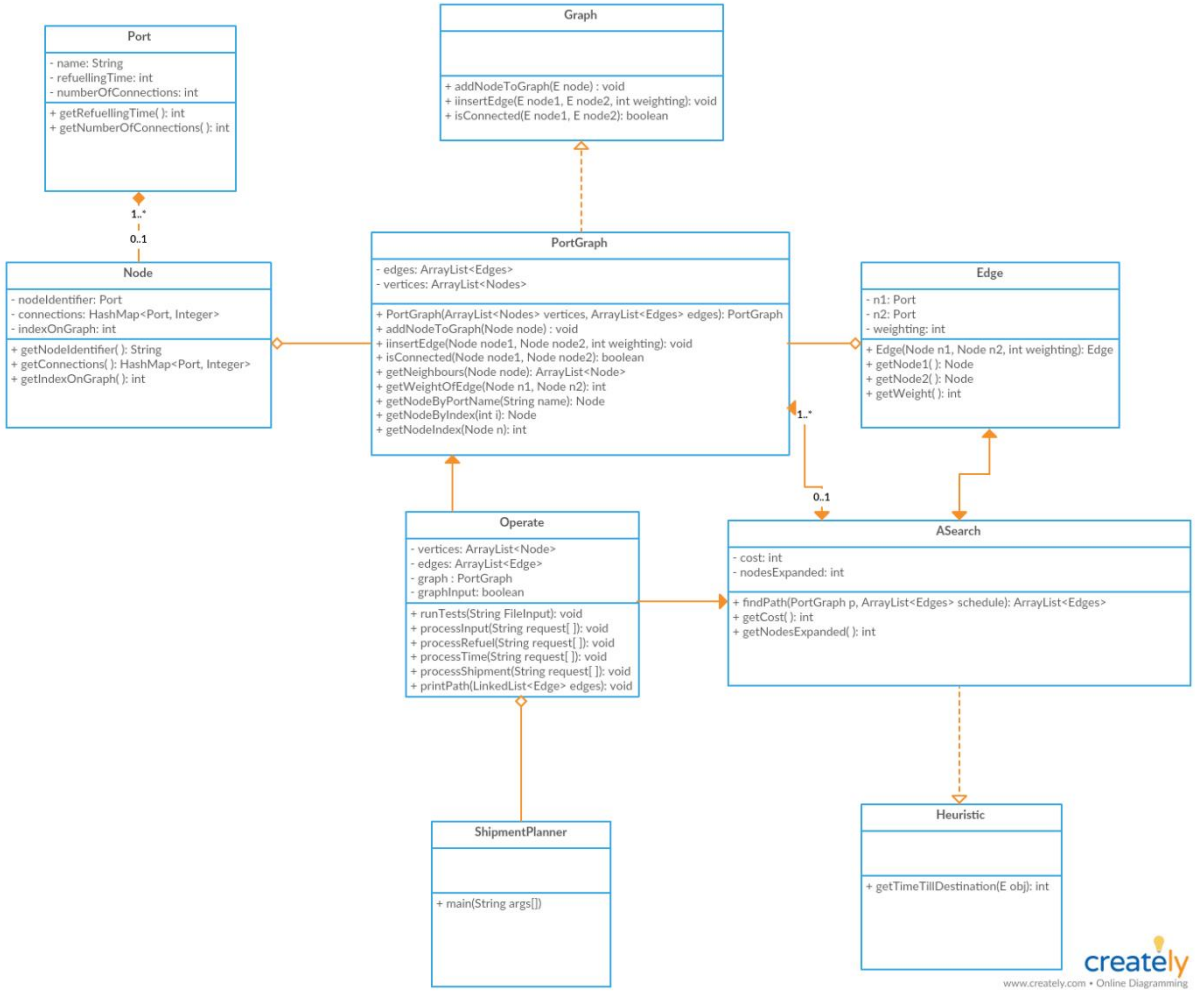
Table of Contents

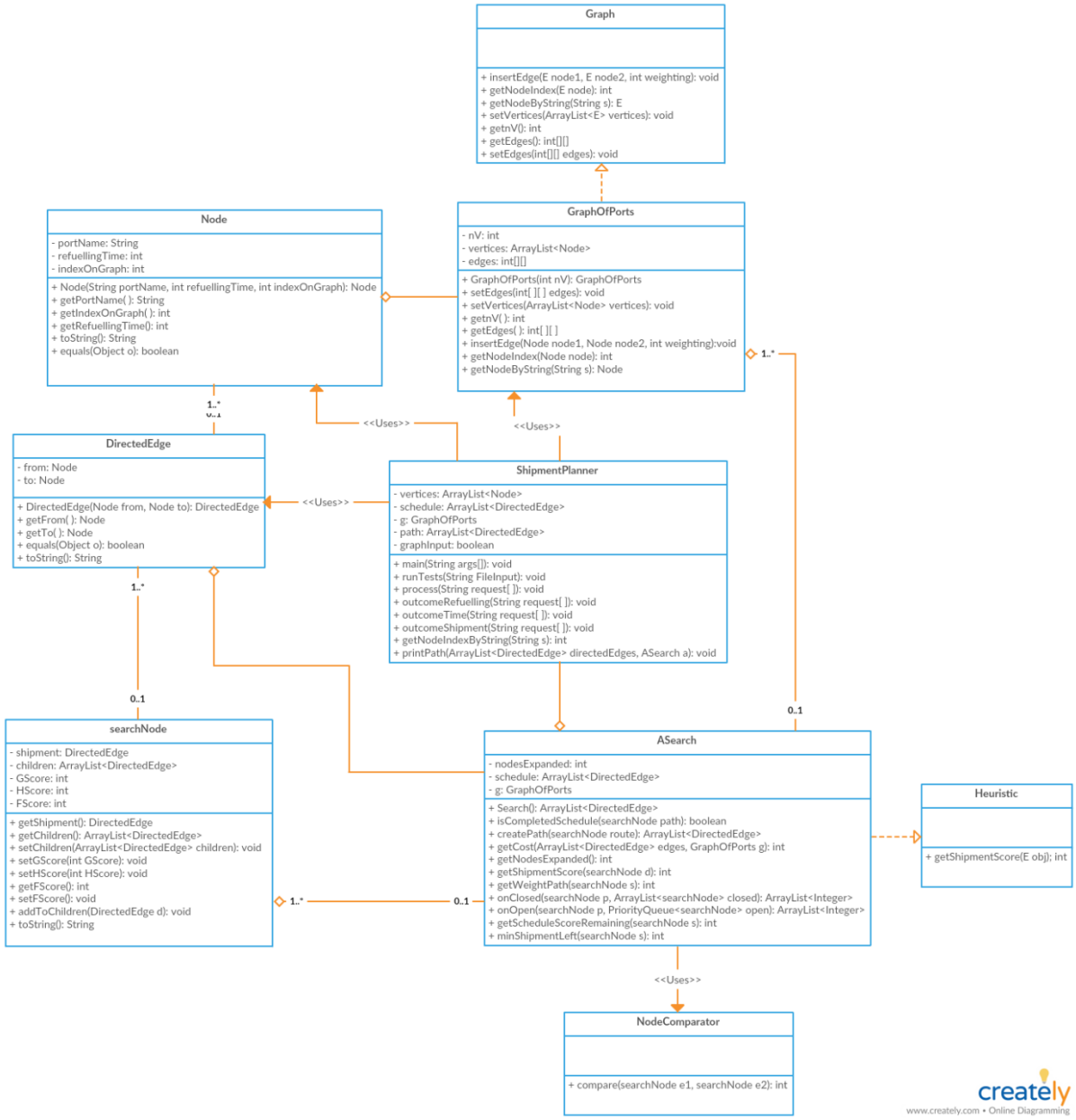
Initial Diagram and Design

This UML diagram is the first one I made before I started any coding, I tried using this exact design however ran into some problems following it the final UML designed after actually coding it was completely different as this ended up being a greedy first search and the design I followed ended up not implementing A*. in turn I had to redesign a majority of the initial design and also recode the entire project to follow A* properly



Final UML class diagram

When I had finished the code the UML diagram and design ended up being entirely different, when implementing A* I realised I needed to be able to follow multiple paths of shipments at once. To do this I added in a new class of search Node, which is essentially a path. I also decided to implement my heuristic to become admissible and also be implemented in the A* search itself rather than the comparator. I also decided to bootstrap my shipment planner class rather than do the operate class as shown above and in the previous assignment and implement the data processing, so multiple instance of shipment planner can be called. This re-design was based on the fact I did not follow A* search and tried to do it with a greedy search, and overall code re-design to correctly implement the problem at hand



Project Design

Walkthroughs

For Refuelling Request

1. ShipmentPlanner class will read input line by line
2. ShipmentPlanner will separate the comment from the request
3. ShipmentPlanner will read the first word and check if it is “Refuelling”
4. If it isn’t go to the other 2 checks, or skip to next line
5. ShipmentPlanner class will check if the input is valid
6. ShipmentPlanner class will then call on the Node class to create a node with the request input
7. Node class will use the refuel time, and the string name of the port to create a Node instantized with that input
8. ShipmentPlanner class will then add that node to the list of vertices’

For Time Request

1. ShipmentPlanner class will read input line by line
2. ShipmentPlanner will separate the comment from the request
3. ShipmentPlanner will read the first word and check if it is “Time”
4. If it isn’t go to the other checks, or skip to next line
5. ShipmentPlanner class will check if the input is valid
6. ShipmentPlanner class will then check if the graph flag is false
7. If the graph flag is false, it will create a new graph with size vertex list size and set the flag to be true
8. The graph class will then insert an edge between the two nodes scanned from input and the weight of the edge

For Shipment Request

1. ShipmentPlanner class will read input line by line
2. ShipmentPlanner will separate the comment from the request
3. ShipmentPlanner will read the first word and check if it is “Shipment”
4. If it isn’t skip to next line
5. ShipmentPlanner class will check if the input is valid
6. The directed edge class will create a directed edge from the FROM node to the TO node scanned from input
7. This directed edge will then be stored in the schedule.

For A* search

1. ShipmentPlanner class will call the A* search from the previously scanned input
2. ShipmentPlanner class will then pass the graph and schedule into the A* class
3. A* search will perform the A* search given the graph and schedule
4. A* search class will perform the A* search
5. ShipmentPlanner class will now print out the path resultant from the A* search

Explaining how the graph is created

The graph we are creating will have a list of vertices, this will allow for the graph to allow for identification for the node. The graph will be created when it has a list of vertices and it will use the size of that list as the number of nodes. It will create a 2d matrix with list size and initialise the matrix with -1 to show no new connections and a diagonal of 0 (since the distance from a node to itself = 0). This graph will then be able to create edges by setting matrix values based on node index's

Explaining how the A* search works and search Nodes structure

The A* search will be using search nodes which are my implementation of a path. The A* search will compare these paths in order to find the best path. A search node is essentially a path that contains a parent in our case Sydney->Sydney will be the parent of all paths, the root of all paths. The reason I use Sydney->Sydney is because I work with shipments rather than nodes, in order to use Sydney as the beginning of all shipments I bootstrap Sydney with itself to add it as a parent to all paths. The children to a search node are all the shipments which compose the path from the parent. In the search shipments are added to the children creating a new search node (path) with a different state. This is essentially how the A* search will create multiple paths to compare.

Explaining how the goal state is reached

The goal state is reached when the constructed path from the search node contains ALL the shipments on schedule. Once this condition is met the path is constructed from the search node and returned.

Explaining how the A*search knows if a path is on closed or on open

The A* search will know if a path is on open or on closed by comparing the search node with all the other search nodes on open and on closed. If another search node contains the same shipments as the search node we are comparing with but in a different order
For example
Sydney -> Sydney [Shanghai -> Manila, Singapore -> Vancouver]
Sydney-> Sydney [Singapore->Vancouver, Shanghai -> Manila]
These two paths are in the same state as both contain the exact same shipments. However, these two paths have a different cost, the first expanded path is
Sydney->shanghai->manila->Singapore->Vancouver
The second expanded path is
Sydney->Singapore->Vancouver->Shanghai->manila
These two paths may be the same state, however have very different costs due to refuelling times AND edge weights when connecting paths. The order makes a difference. This difference in weighting allows us to remove certain paths from closed or remove certain paths from open. This is also another way we are comparing multiple search paths in order to find the path with the lowest cost as quickly as possible.

Why my heuristic is admissible

The heuristic I chose is admissible because it is adding the remaining path DIRECTLY along with the closest distance to the nearest shipment. THIS WILL BE ADMISSIBLE because in order to actually reach goal state it must at least go to any of its nearest shipments then complete the schedule, however it is under estimating because it is not linking the schedule, it is just adding the raw weights left if, Sydney -> Vancouver and manila -> Singapore was left it would not connect Vancouver -> manila it would only just add the raw weight to get from Sydney -> Vancouver and manila -> Singapore, it would not account for linking, and Since our path has to choose between Sydney or manila as its next connection it will pick the distance closer, so if Sydney is closer to our node than manila it would chose manila because the path must connect to at least one of the nodes. This means that the heuristic will always be admissible as we are always underestimating the cost of the path.

How did I implement the strategy pattern for the heuristic

I had implemented a heuristic interface which can be implemented on any class or any other language, it just contained the method which the heuristic used to score the estimate of score remaining till goal is reached. The ASearch class implements the heuristic interface and has its own method of implementing the estimate score, in my case I used shipments remaining + closest distance to next shipment, however since this was supplied through the heuristic interface if the problem was different I could of for say taken the Manhattan distance, or another method depending on the requirement. this is how I made use of the strategy pattern.

CRC Cards

ASearch

The A* search class is my class which will utilise all other classes in order to bring the project to completion. It will perform the search, calculate the costs and return the final path if one exists to the optimal path that completes the schedule which is a list of directed edges

Responsibility	Collaborator
- Will score a search node's actual path based on its current path - Will perform the heuristic scoring on a search node based on its current path - Will perform the A* search and either return a path if one exists, or return null - Will check if a search node has reached goal state	- DirectedEdge - SearchNode - GraphOfPorts - Node (only for summing costs) - ShipmentPlanner - NodeComparator

DirectedEdge

This class represents a shipment with direction e.g. From Sydney -> manila, from node = Sydney, to node = manila. This class is reliant on the node class as without the node class it cannot exists. It is composed of a node pair with direction

Responsibility	Collaborator
- Will create a connected edge with direction given a FROM port and a destination TO port. - Will be able to compare other directed edges (shipments)	- Node - searchNode

Node

The node class will form the most basic class which is representative of a port. A shipment is a movement from a port to a port. This will contain basic information for a port such as name, refuel time and also the index on the graph for the port (for search ease)

Responsibility	Collaborator
- creates ports with a name/refuel time/index on graph - will be core of graph (allows you to retrieve node information and give nodes identity on graph)	- GraphOfPorts - DirectedEdge

GraphOfPorts

The GraphOfPorts class is designed to be an adjacency matrix graph class implemented from the generic graph interface. It will be used to add weights between nodes and required to perform the search.

Responsibility	Collaborator
- This class is responsible for creating the graph during the search - This class is responsible for assigning weights between to nodes in the graph - This class is responsible for returning the weight between two nodes for calculating cost (used by A* class)	- Node - ShipmentPlanner - A*Search

NodeComparator

The NodeComparator class will be used as a comparator for the priority queue ranking searchNode's based on their FScore.

Responsibility	Collaborator
- Priority queue comparator, will be comparing search nodes based on their FScore's - Will order elements in the priority queue	- ASearch - searchNode

searchNode

The searchNode class will be used as a represented state in the A*Search. Paths will all originate from the root Sydney->Sydney (since we are working in shipments) and a list of shipments (children) that compose the path. This will be the "node" used in the search since we are comparing states of paths not nodes themselves.

Responsibility	Collaborator
- this class is responsible for creating "paths" (list of shipments with a root) - retrieve path scores - adds a shipment to path - creates other paths based on current path as well	- DirectedEdge - searchNode

ShipmentPlanner

The ShipmentPlanner class will be the main class which will read file input passed in the command line, and then process the input according to the requests in the file, and after it has finished reading the file and passing all data in to create the graph and schedule, it will call the A* search to perform the A*search, and after it is complete it will print the respective path (nothing if no path is found).

Responsibility	Collaborator
- will be passing data for graph creation - passes data for schedule creation - calls the A*search to find path based on that schedule - print out the return from the A*search	- GraphOfPorts - Node - DirectedEdge - ASearch

Run-time complexity analysis

(as quoted from Wikipedia)
Consider integer n which represents the number of shipments where n>=0
And consider integer m which represents the number of nodes on graph where m >=0 (outdegree of all nodes are m since they are all connected)
The runtime complexity analysis for the A*Search without a heuristic in worst case is O(m^n). in best case it will be O(n) where n is the number of shipments on the schedule. However, through the use of my heuristic I will be eliminating paths that are “useless” and prioritising paths which have great scores. The way my heuristic will do this is take the path, and add the remaining schedule weight to the path, and the closest distance to the nearest node on shipment to the last node in the path (0 if the last node in the path is a shipment itself). What this will do is, paths which have longer distance to complete shipment, or paths which are further away from shipments are discarded, their scores will be high and towards the end of the open set. This will get rid of paths which have longer distances to reach goal state since they will not be optimal paths. In order to calculate the effectiveness of the heuristic I will be using the formula supplied on Wikipedia that
N + 1 = 1 + k + k^2 + …. + k^n to solve for K, where N are the number of nodes expanded. In one of my searches 19 nodes expanded, and n = 7 m = 5
20 = 1 + k + k^2 + k^3 + k^4 + k^5 + k^6 + k^7
19 = k + k^2 + k^3 + k^4 + k^5 + k^6 + k^7
Using an online calculator
K = 1.25215
The heuristic I have used brings runtime costs to O(1.25215^n) rather than the O(5^n). this is because the heuristic will get rid of all extra branching which will result in less suboptimal paths checked, since suboptimal paths will not be expanded on further during search.