ARTIFICIAL INTELLIGENCE FINAL PROJECT

INTRODUCTION

The objective of this project was to develop and compare various search algorithms in a simulated pizza delivery map without explicitly defining a solution. A single driver was initialized and calibrated to carry a maximum load of pizzas. The map itself consisted of various terrain tiles, ranging from pizza shops, customers, obstacles, and pathways of varying cost. On tiles with pizza shops, the driver could load pizzas one-by-one into his bag, and on tiles with customers, the driver could drop pizzas one-by-one, decreasing the number of pizzas in the customer queue. By searching the map using heuristics and cost analysis, the simulation could produce the optimal solution, or in other words, the solution with the mininum number of required steps, to deliver all the pizzas.

The development of the problem itself was split into two parts, the basic problem and the advanced problem. The basic problem limited the terrain type to customers, shops, buildings, and streets, and compared varying maps with at most 2 pizzas per customer.

The advanced problem utilized a more complicated map, with terrain of varying cost and customers ranging from orders of 1 to 3 pizzas. By splitting the task into two parts, I was able to focus firstly on the effectiveness of the algorithm itself and secondly on the admissibility of the heuristic equations.

Part I:

BASIC PROBLEM

DESIGN & IMPLEMENTATION

Problem Representation

State	The state is a 3-element tuple as follows:
	`(driver, load, customers)`
	
	Driver is a two element tuple: `(x, y)`
	
	Load is an int, the number of pizzas in the bag.
	
	Customers is a tuple of 3-element tuples. A tuple is used instead of a list so that the data is immutable. Each element represents a customer, and tracks the customer's location and the number of pizzas left to deliver. Each element inside of the customers tuple is as follows:
	`(customerX, customerY, pizzas)`
	
	Customers with 0 pizzas left to deliver are not included in the list.

Actions	The actions can be separated into two categories: movement-related and pizza-related. Movement-related actions: *North*, *South, East*, and *West*. As implied, these actions move the driver in the direction specified, with South being the +y direction and East being the +x direction. Pizza-related actions: *Drop* and *Load*. *Drop* decreases the pizzas in the driver's bag and *Load* increases it.

Implemented Functions

`actions()`	This function receives a state and returns the valid actions. For example, it will not allow a movement-related action that will send the driver into a building or out of bounds. It will also not allow the driver to load more pizzas than his capacity, and will only allow dropping at customer tiles with deliveries remaining.
`result()`	This generates and returns the updated state given the current state and an action.
`is_goal()`	This compares the current state with the goal state and returns whether or not they are equivalent.
`setup()`	This initializes the state given the configuration file. It also defines the final state, which by default is the state where the driver is in his original tile with no pizzas in his load and no customers remaining.
`printState()`	This simply returns a human-readable string to display the current state variable.
`getPending Requests()`	This reads the current state of customers and compares it with the initial state to determine how many pizzas are left to deliver. This is used to display the correct image.

PERFORMED TESTS

Map #1

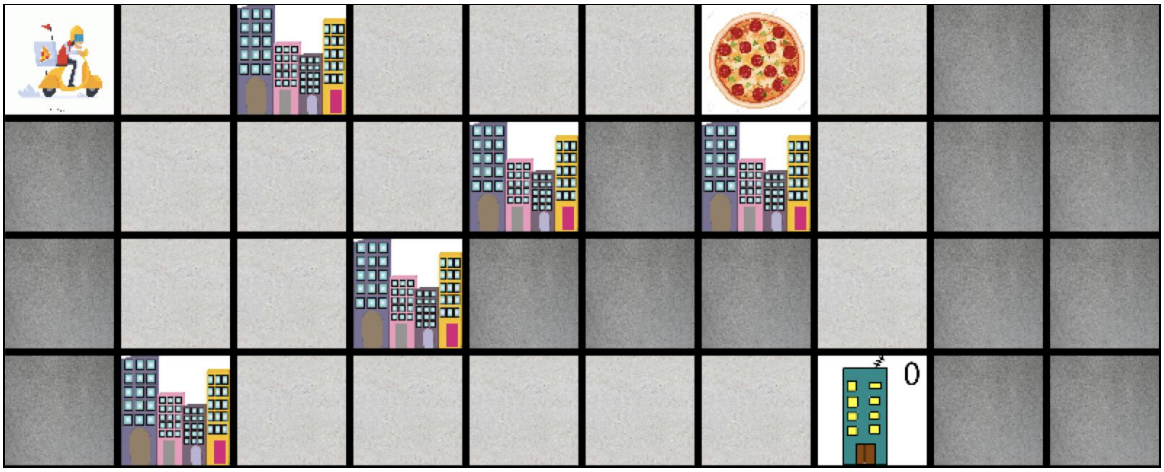

Figure 1: Map 1

This simple and traditional map provided multiple possible solutions with a simple objective: one customer with two pizzas. The test results are shown in the table below. As expected, the A* and Breadth First search performed very similarly, with breadth first finding the solution in slightly less iterations.

		Total Length	Total Cost	Visited Nodes	Iterations	Max Fringe Size
Map 1	A*	26	26	279	279	25
	BF	26	26	271	271	23
	DF	46	46	122	122	42

Table 1: Map 1 Test Results

Map #2

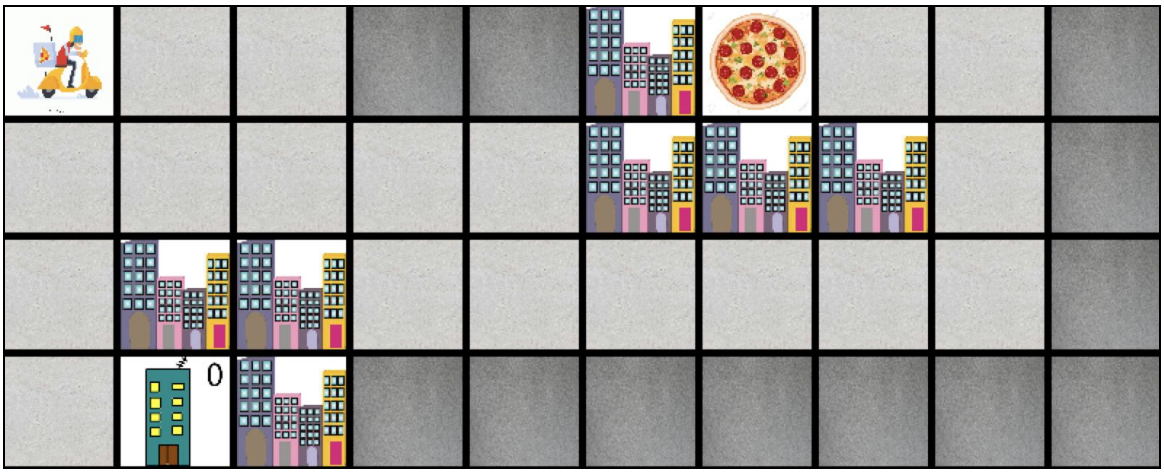

Figure 2: Map 2

This slightly more complex map intentionally routed the driver to one side of the map to the pizza shop. There was effectively only one possible order of delivering the pizzas. The test results are shown in the table below. As expected, A* and Breadth First performed best.

		Total Length	Total Cost	Visited Nodes	Iterations	Max Fringe Size
Map 2	A*	36	36	107	107	7
	BF	36	36	107	107	7
	DF	46	46	145	145	43

Table 2: Map 2 Test Results

Map #3

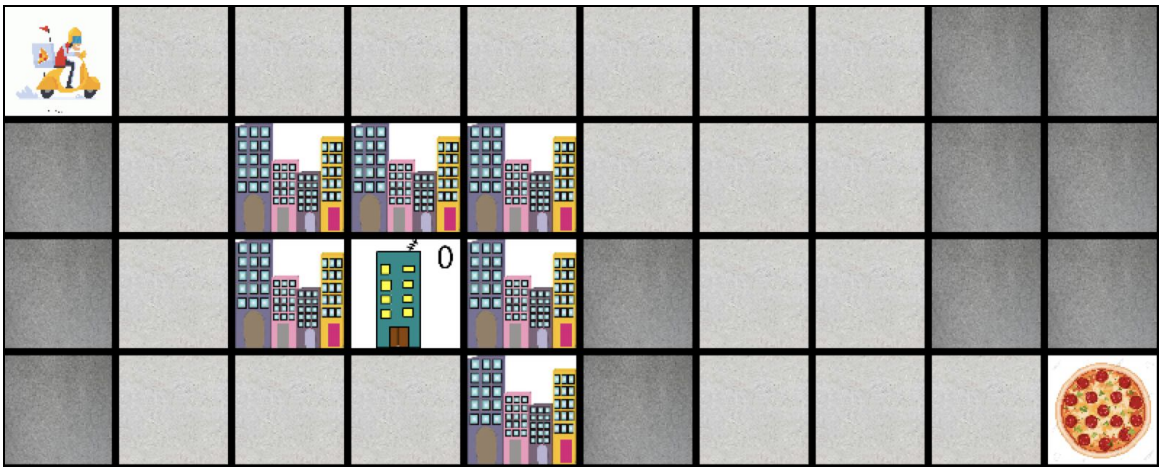

Figure 3: Map 3

The third map, the most difficult to solve, tested the effectiveness of the algorithm when intrusive obstacles are placed around the customers. It is interesting to note that the driver takes a different, equally efficient, path to return from the pizza store than it did to reach the pizza store. The data is shown in the table below. Once again, as expected, the most efficient algorithms were A* and Breadth First Search, with Breadth First operating on a slightly smaller fringe size.

		Total Length	Total Cost	Visited Nodes	Iterations	Max Fringe Size
Map 3	A*	40	40	140	140	10
	BF	40	40	140	140	8
	DF	46	46	147	147	51

Table 3: Map 3 Test Results

Part II:

ADVANCED PROBLEM

DESIGN & IMPLEMENTATION

Problem Representation

Modifications	The map size was increased to the largest reasonable size, and a complex map was designed where many solutions were possible. This allowed increase emphasis on the heuristic functions as a means of determining efficiency.
New Terrains	Two new terrain types were implemented: Field and Hill Field terrain has a cost of 2 to traverse, and hill terrain has a cost of 3 to traverse.

Implemented Functions

`cost()`	The cost function is given an action and a state and returns the cost to perform the action. In the advanced problem, the cost is *not* universal, so the function extracts the cost attribute from the configuration file for the given terrain type. For loading and dropping, the cost is 1 for each pizza loaded or dropped respectively.
`heuristic()`	This function is given the current state and generates a number representing the “minimum” effort required to solve the problem. As described below, 4 different heuristic equations were tested and compared. It is important that the heuristics are all admissible, meaning that they do not *over*estimate or make assumptions based on the limited information available. In this case, less informed is better.

TESTS PERFORMED

The same map and cost function were used in all of the heuristic tests in order to compare the effect the heuristic function alone had on the efficiency of the solution. The four heuristic functions, along with a zero-heuristic example, are separated into 5 unique folders, “Heuristic 1”, “Heuristic 2”, “Heuristic 3”, “Heuristic 4”, and “No Heuristic” respectively. As visible in the map below, the dynamic arrangement of the terrain means that there are multiple possible solutions: it is up to the heuristic function to define the best.

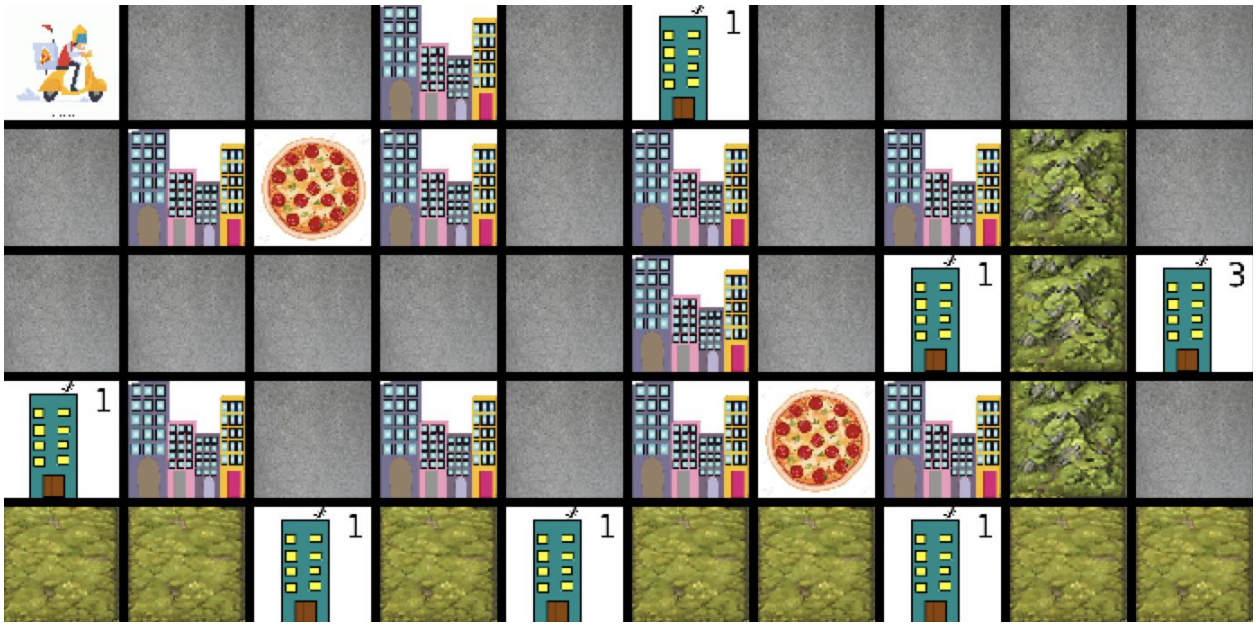

Figure 4: Advanced Map

Heuristic #1: Pizzas Remaining

The first heuristic equation focused on the number of pizzas remaining to deliver in a given state. The algorithm simply summed the total pizzas yet to be delivered, ignoring the location of the customers.

```
heuristic = pizzasRemaining = sum([c[2] for c in state[2]])
```

Heuristic #2: Customers Remaining

The second heuristic equation focused on the number of customers remaining in a given state. The equation simply totaled the number of customers still in the queue, disregarding how many pizzas were remaining to be delivered.

```
heuristic = customersRemaining = sum([1 for c in state[2]])
```

Heuristic #3: Pickup Trips Remaining

The third heuristic function attempted to calculate how many pickups and dropoffs were remaining in total. To do this, the algorithm subtracted the number of pizzas in the load from the number of pizzas remaining and divided the number by MAXBAG (for every full bag of pizzas, another trip to the pizza shop must be performed).

```
pizzasRemaining = sum([c[2] for c in state[2]])
heuristic = (pizzasRemaining - state[1]) / self.MAXBAGS
```

Heuristic #4: Manhattan Distance

The final heuristic function utilized the traditional manhattan distance function, defining the heuristic as the total vertical and horizontal distance from the final goal state. This is less practical in this particular problem as there are multiple “sub-goals” that alternate continuously throughout the delivery.

```
xDistance = abs(state[0][0] - self.GOAL[0][0])
yDistance = abs(state[0][1] - self.GOAL[0][1])
heuristic = xDistance + yDistance
```

Heuristic Test Results

As shown in Table 4 below, the heuristics have only a small effect in the overall efficiency of the algorithm. All heuristic equations found the same efficient solution, but Heuristic 4 managed to find it in slightly less iterations and the smallest overall fringe size. It is worth noting, that although “No Heuristic” was technically the least efficient, it still found the optimal solution.

	Total Length	Total Cost	Visited Nodes	Iterations	Max Fringe Size
Heuristic 4	64	77	29709	29709	1165
Heuristic 2	64	77	30491	30491	1171
Heuristic 1	64	77	30499	30499	1274
Heuristic 3	64	77	30534	30534	1203
No Heuristic	64	77	30564	30564	1146

Table 4: Advance A* Results (Sorted by Efficiency☆)

As shown in Table 5, the greedy algorithm showed starker contrast between different heuristic equations. Heuristic 4, the manhattan heuristic, was the least efficient by far. As expected, none of the heuristics found a solution as optimal as the one found by A*.

	Total Length	Total Cost	Visited Nodes	Iterations	Max Fringe Size
Heuristic 3	98	129	456	456	187
Heuristic 1	100	116	455	455	132
Heuristic 2	116	139	524	524	149
No Heuristic	132	165	1081	1081	432
Heuristic 4	176	200	12624	12624	1172

Table 5: Advance Greedy Results (Sorted by Efficiency☆)

☆ *Note: In the context of these tests, “efficiency” prioritizes Total Length, Total Cost, Visited Nodes/Iterations, and Max Fringe Size in that order. The results are sorted accordingly.*

CONCLUSION

For the basic maps, the conclusion is relatively consistent across all map arrangements: as highlighted yellow in Table 1 through Table 3, the most efficient algorithm for all categories was Breadth First search, with A* algorithm placing in a close second. More specifically, A* and Breadth First Search found solutions with the same amount of steps, but Breadth First Search traversed the solutions with less iterations and a smaller max fringe size. Alternatively, Depth First Search failed to find the most efficient solution given the basic map arrangements. Overall, this performance aligns with the theory taught in this course.

For the advanced maps, the effect of the heuristic equations varied greatly depending on the search algorithm used. When using the A* algorithm, the most efficient solution was found every time, but there were slight variances in the number of iterations given the heuristic functions. No heuristic was the least efficient in terms of iterations, but still found the most optimal solution. The greedy algorithm, on the other hand, found dramatically different solutions when given different heuristic equations. The most efficient heuristic was Heuristic 3, which attempted to quantify the number of pickup trips remaining in the delivery. The only heuristic less efficient than no heuristic was Heuristic 4, the manhattan distance heuristic. As explained above, this aligns with expectations as the “goal location” changes throughout the search as pizzas are delivered and customers are eliminated.

PERSONAL COMMENTS

One interesting challenge when developing this project was accounting for the effects of scaling. As the maps became bigger and more complex, it became exponentially slower to calculate a valid solution. In an ideal experiment, I would have liked to run tests on significantly bigger and more complex maps, but this was unrealistic due to constraints of time and computing resources.

Overall, I think the largest benefit of this project was gaining hands-on experience with the different search algorithms we had previously been calculating by hand. To apply our knowledge and generate a solution without hard-coding a path opens a whole new world of opportunities in my career as a computer engineer.