



Universidad  
Carlos III de Madrid

# ARTIFICIAL INTELLIGENCE

## Final Project Assignment

ROUTE SEARCH IN LOGISTICS PROBLEMS

GROUP 88

AUTHORS:

Rafael Pablos Sarabia (100372175) [100372175@alumnos.uc3m.es](mailto:100372175@alumnos.uc3m.es)

Juan Sánchez Esquivel (100383422) [100383422@alumnos.uc3m.es](mailto:100383422@alumnos.uc3m.es)

## TABLE OF CONTENTS

|   |    |
|---|----|
| 1. INTRODUCTION .....                               | 2  |
| 2. PART I – BASIC PROBLEM .....                     | 2  |
| 2.1 DESIGN.....                                     | 2  |
| 2.2 IMPLEMENTATION .....                            | 3  |
| <i>setup(self)</i> .....                            | 3  |
| <i>actions(self, state)</i> .....                   | 4  |
| <i>result(self, state, action)</i> .....            | 4  |
| <i>is_goal(self, state)</i> .....                   | 5  |
| <i>cost(self)</i> .....                             | 5  |
| <i>heuristic(self, state)</i> .....                 | 5  |
| <i>printState(self, state)</i> .....                | 5  |
| <i>getPendingRequests(self, state)</i> .....        | 5  |
| 2.3 EVALUATION.....                                 | 6  |
| 2.4 COMPARISONS .....                               | 10 |
| 3. PART II – ADVANCED PROBLEM .....                 | 11 |
| 3.1 UPDATED COST(SELF).....                         | 11 |
| 3.2 UPDATE HEURISTIC(SELF, STATE) .....             | 11 |
| 3.3 TEST FOR MULTIPLE CLIENTS AND RESTAURANTS ..... | 12 |
| 3.4 MULTIPLE TYPES OF TERRAINS AND COSTS .....      | 14 |
| 3.5 COMPARISON.....                                 | 17 |
| 4. CONCLUSION .....                                 | 19 |
| 5. PERSONAL COMMENTS .....                          | 19 |

## 1. Introduction

The purpose of this final project is to design and implement the functions needed for using the SimpleAI library in Python, which contains different search algorithm, and analyze the different results obtained depending on the problem and the search algorithm.

The problem is focused on finding a route in a logistics problem where a worker has to deliver pizzas to different customers on the grid. For the basic part of this project, the problem is simplified so that there is only one client and one restaurant from where the delivery person can stock up on pizzas. Also, the client will order two pizzas at most, which is the maximum number of pizzas that the worker can carry on the bike and the cost of each action will always be 1. For the advanced part, more parameters can be configured such as costs for different tiles, number of customers and pizzas ordered, number of restaurants and maximum number of pizzas in the bike. In both parts, the delivery person must start at the base and return there once all pizzas have been delivered.

## 2. Part I – Basic Problem

This section of the project consists on developing the functions needed for the SimpleAI library to be able to execute. In order to simplify the project at the beginning, several constraints are set: one restaurant, one client with two pizzas ordered at most, maximum load capacity of the delivery person's bike is 2 pizzas, deliver person cannot ride through buildings, and all actions have unit-cost. After the design and implementation of the functions, we had to perform an experimental evaluation and comparison of the different results for different search algorithms and initial states. In this first part, many of the implemented functions have been coded thinking about the next section. Therefore, some functionality included (such as load or customers with 3 pending pizzas) was added even though it was not required for this section.

### 2.1 Design

We first had to think about the problem representation and how to organize the information needed. Then, we had to detail the design and functionality of each function. Next, the main design decision taken are explained.

State Representation: states need to have information about the current position of the delivery worker, the position of the customers with one, two or three pending pizzas, and the number of pizzas currently loaded in the bike of the delivery worker. Therefore, state will have the following format:

$State = (< Position (x, y) >, < Customer1 >, < Customer2 >, < Customer3 >, < PizzasLoaded >)$

$CustomerN = ((x_1, y_1), \dots, (x_m, y_m))$ ,

with  $x_1, y_1, \dots, x_m$  and  $y_m$  being positions of customers with  $N$  undelivered pizzas

Initial State: the initial state will have the starting position of the agent, the initial clients in each corresponding tuple based on the undelivered pizzas, and 0 for the initial amount of pizzas loaded in the bike.

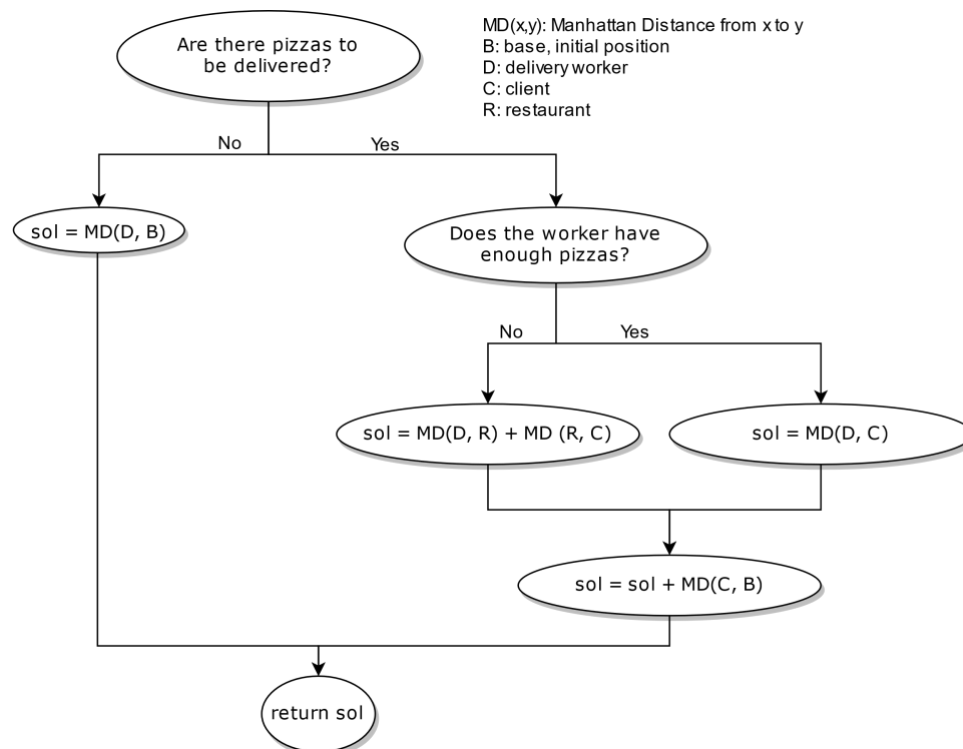
Final State: the final state will have the end position of the delivery worker (which must be the initial one), empty customer tuples since all pizzas must be delivered, and 0 as the amount of loaded pizzas in the bike.

Variables: in addition to the variables initialized by the initially provided functions, we have assigned values to variables MAXBAGS (maximum amount of pizzas in the bike), SHOPS (position of every restaurant available), CUSTOMER (list of lists with customers with the same initial number of pending pizzas) and ALGORITHM (string to choose the search algorithm to use).

Possible Actions: the possible actions are the different directions in which the delivery worker can move or load and unload. It is important to note the order in which they are obtained since it will influence the output in algorithms such as depth first search. The actions are: West, North, East, South, Load and Unload. The results of these actions will influence the next state by changing the current position, the amount of pizzas in the bike or the remaining customers.

Cost: all actions will have unit cost (movement, load and unload).

Heuristic: for this first part, the Manhattan Distance is an appropriate heuristic to use since we are in a grid with 4 directions of movement. However, we will need to consider the different steps that the delivery worker must follow since specific positions are mandatory. The logic for the heuristic is explained below:



## 2.2 Implementation

On this part of the project, after reading both the statement and the provided code several times and having clear the goals of the code, we had to develop the code itself. Starting was the hardest part as soon as something was modified, the game would stop working. Once we figured it out for a simple case, introducing constraints, conditions and information into the state was simpler. On each of the following subcategories the functions implemented by us are explained in detail.

### setup(self)

Inside this function we will first create customers and shops lists in which we will store as the name states the customers and the shops for our given map allowing us to have easier access to them in the future. This has been done because they are commonly used through the code. Also, the maximum load is saved in a variable for easier access too.

In addition to those lists, we declared and initialize the initial and final states of our problem. They both will be tuples containing as elements other tuples or numbers, the tuples that they contain will also contain either tuples or numbers. Bellow you can see an image and the explanation for each of the elements in our states.

```

-- Initial State --
((0, 0), ((7, 0),), ((4, 3), (7, 2), (9, 3)), (), 0)
-- Final State --
((0, 0), (), (), (), 0)

```

**Pizza deliverer coordinates:** In this first tuple the x and y starting coordinates of the pizza deliverer will be stored. The final state coordinates for the pizza deliverer should be the same as the starting ones.

**Clients with a one pizza order coordinates:** This second tuple contains a tuple for each client coordinates who have just one pizza remaining. At each delivery, the client's coordinates will be deleted from the state or moved to another tuple, so that the tuple should be empty at the end.

**Clients with a two pizzas order coordinates:** This third tuple contains a tuple for each client coordinates who have two pizzas remaining. At each delivery, the client's coordinates will be deleted from the state or moved to another tuple, so that the tuple should be empty at the end.

**Clients with a three pizzas order coordinates:** This fourth tuple contains a tuple for each client coordinates who have three pizzas remaining. At each delivery, the client's coordinates will be deleted from the state or moved to another tuple, so that the tuple should be empty at the end.

**Number of pizzas being carried:** This last element of the tuple will contain the number of pizzas currently being carried by the pizza deliverer. It can be 0, 1 or 2 (2 will be the maximum capacity of the motorbike for this part). At the end, as it can be guessed, no pizzas should remain on the motorbike.

Lastly, we also specified the desired algorithm to use for the search problem. To accomplish that objective, we declared a variable called `ALGORITHM` in which the user will specify the desired algorithm to be used (among those we are implementing) and then we will use that information inside this function for selecting the proper algorithm.

#### actions(self, state)

The objective of this function will be to return the possible actions that our pizza deliverer can do given its state. To do so, we create an empty list to which we will append the possible actions if the conditions are met. For movement, those conditions will be that the cell is in range (within the limit of the given map) and that the cell is not a building (as we cannot go through them). If both conditions are met, the checked direction (North, South, East or West) will be appended to the list. Also, if the position is that of a restaurant and the delivery person has not reached the maximum capacity of the bike, another appended action will be Load. Unload will be appended if current position is that of a customer. Lastly, we will return the list with all the possible actions.

#### result(self, state, action)

This function will be the one doing most of the verifications and the one that will influence the most our result as most of our code is here. It will return the next state for our deliverer given a state and an action.

To begin, if the action is regarding movement, we will create a `next_coordinates` tuple in which we will store the next coordinates for our deliverer, they will be based on the ones given through the state plus the action. If action was regarding movement, state with new position will be returned along with current customers and pizzas being carried.

Then, we will check if the deliverer is at a restaurant cell. If so, we will calculate the number of needed pizzas (by checking the number of customers in the state and how many pizzas they need) and pick a number accordingly (bearing in mind we can carry up to MAXBAGS pizzas) so the delivered will pick between 0-MAXBAGS pizzas depending on the needed ones. After determining the number, we will return the next state with the current coordinates and customers and an updated number of pizzas carried.

The other option is for the action to be Unload. In this case, different set of instructions will be executed depending on whether the current position is that of a customer with 1, 2 or 3 pending pizzas. In all cases, the idea is to calculate how many pizzas can be delivered, calculate the remaining pizzas in the bike, and update the customer tuples after the current delivery. We must consider that a customer with 3 pending pizzas may have 0, 1 or 2 pending pizzas after a delivery has been done.

Lastly, just in case that the proper returned statement was not executed, we will return the state where nothing has changed.

#### is\_goal(self, state)

The objective of this function will be to return whether we have ended or not. To do so, we will simply have to check if the given state is the same as our goal state and return that result.

#### cost(self)

This function will return the cost of a given action which will always be 1 for this part. However, for the advanced part, we will include other costs for moving on different terrains.

#### heuristic(self, state)

In this function a proper heuristic function is developed. This heuristic will only be valid for one customer and one pizza restaurant, even though it may work fine in some other specific cases. For a control flow diagram, please refer to the previous section.

The heuristic is based on the Manhattan distance. If there are no pizzas to be delivered (no clients in our client's tuples), the value will be the Manhattan distance from the deliverer's position to its starting point. Else, if there are still clients and we have pizza loaded, we will take the value of the Manhattan from our deliverer to the client plus the distance from the client to the starting point. Else, if there are still clients but the deliverer has no pizza loaded, we will take the Manhattan distance from our deliverer to the nearest restaurant plus the distance from that restaurant to the client plus the distance from the client to the starting point. In every case, a value will be returned. Please note that this heuristic is admissible and consistent for the basic part of the project. For the advanced part, a different heuristic will be developed and explained in the corresponding section.

#### printState(self, state)

The aim of this function is to print in the user console information about the current state. To do so we return in a string the desired information obtained from the given state.

#### getPendingRequests(self, state)

Lastly, this function will return how many pending pizzas the customer in the delivery worker's current position has, if any. This function will be used to determine the corresponding image (with the number of pending pizzas) for each customer cell. None will be returned if we are not on a customer cell. If a cell position is not that of a customer stored in the current state, but was in the initially provided set of customers, all his/her pizzas have been delivered and therefore 0 is returned.

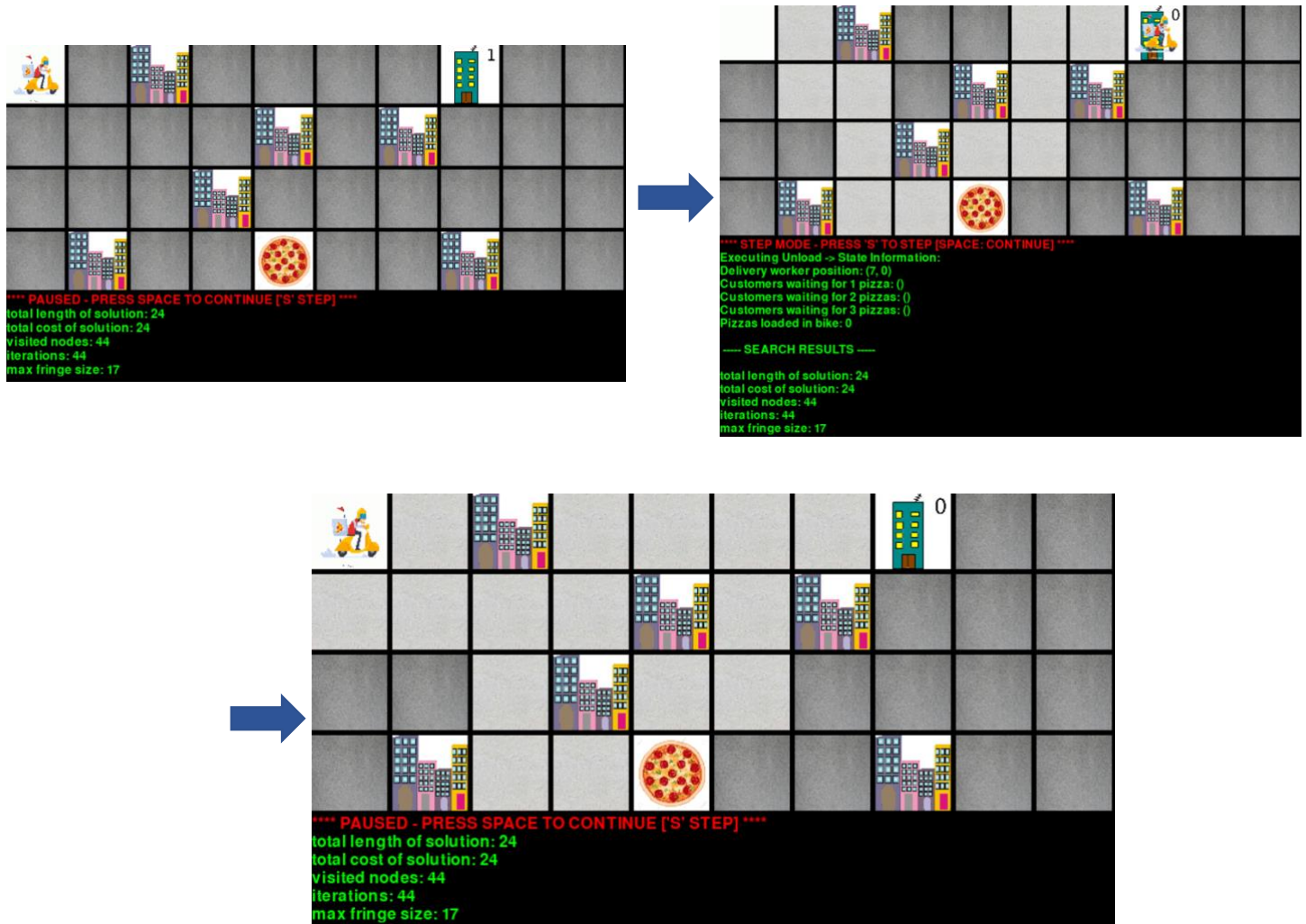
To check that, we will first try to find the current position in any of the tuples of the current state representing customers. If it is found, the value of the tuple for the remaining pizzas is returned. Lastly, if the position cannot be found, we will return zero if the customer was in the initial set of customers as it will mean they already have all the delivered pizzas or None if it was not a customer.

## 2.3 Evaluation

After the code was implemented and we believed it worked as expected, it was time to test its behavior on different environments through different tests. When using A\*, to compare whether our heuristic was admissible or not, we compared with the result the program provided when using the Breadth-First Search algorithm, as it always finds the optimal solution with the least cost. If tests were successful, they will appear on **green**; otherwise they will appear on **red**. More complex test cases with more conditions are made on the advanced problem section.

### TEST-01

Testing for a basic example using A\* algorithm.

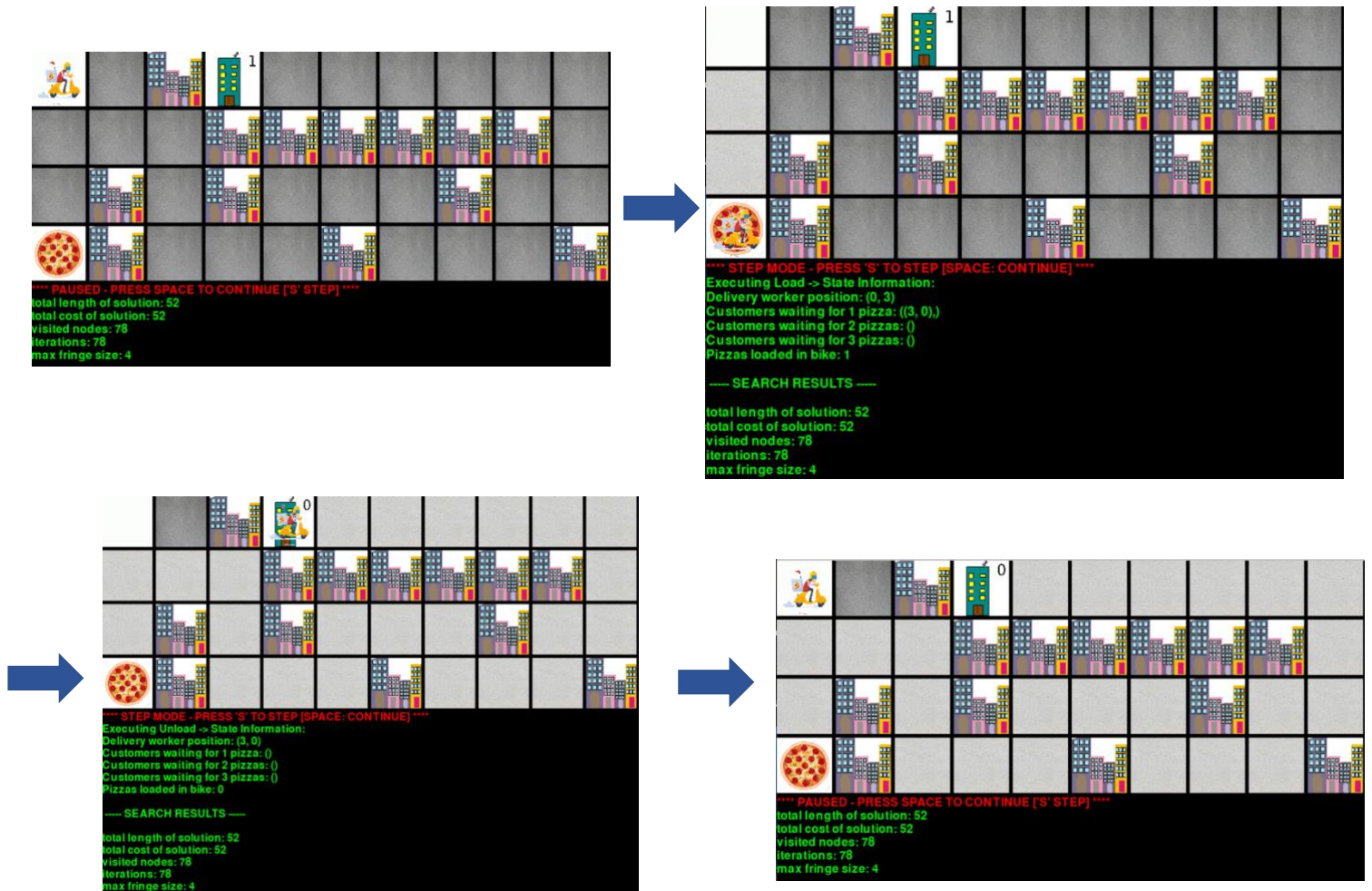


As we can see it behaves as expected, first going to the pizza, then for the client, and lastly going back. We can also observe on image two the status of the current state is printed for each state it goes through accurately. In addition, we observe the number of pending pizzas on customers decreases at it should. Finally, the number of visited nodes is low, which led us to believe that our code is efficient as a greater number of visited nodes would mean worse performance.



## TEST-02

Testing for a complex route example using A\* algorithm.

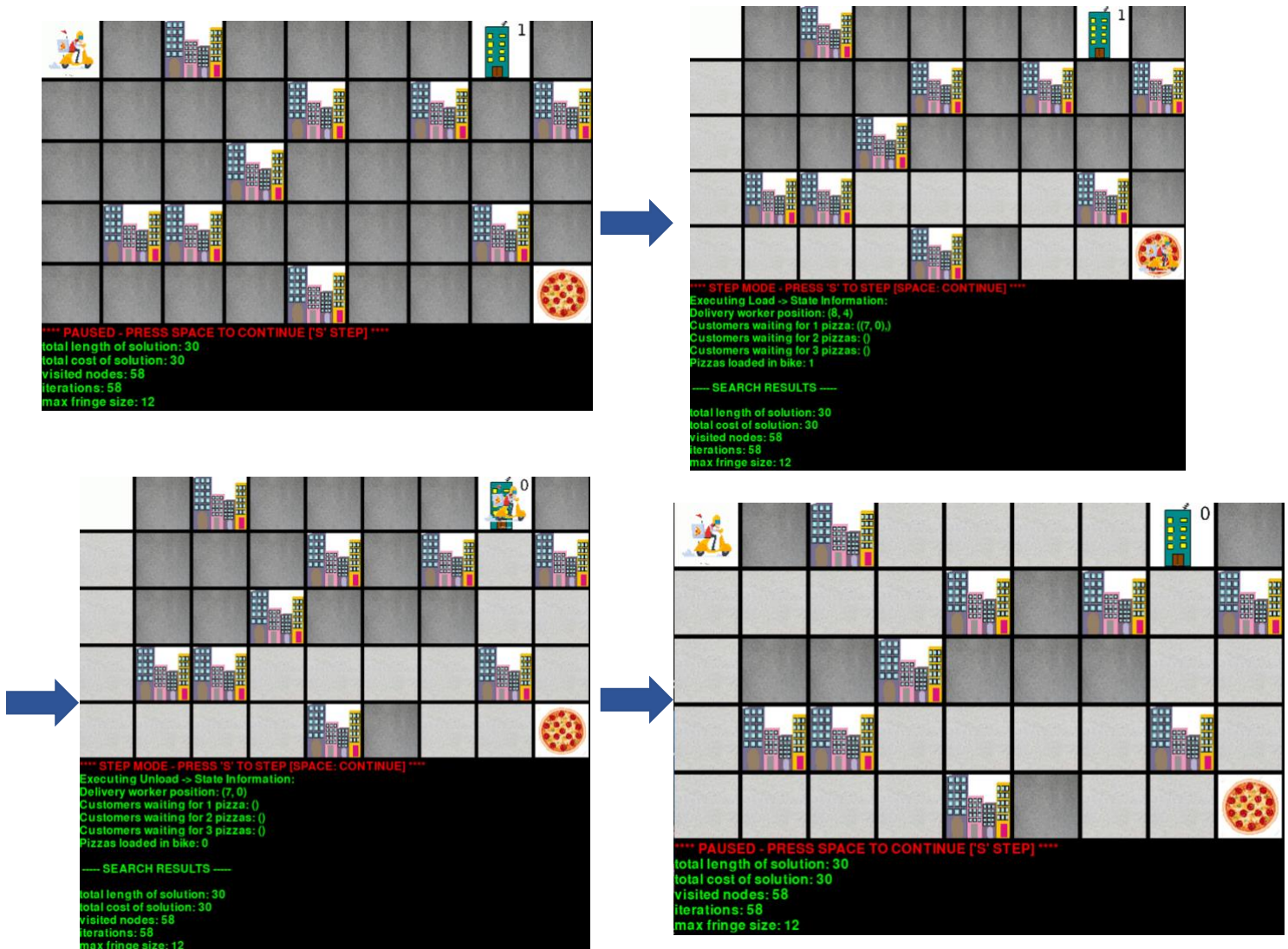


As it can be observed, it behaves as expected by avoiding at all times the buildings and never going through them, first going to the pizza, then delivering it and then coming back. It can also be seen how it only picks the necessary number of pizzas (one) and how it unloads it and removes the customer from our tuple one the delivery has taken place.



### TEST-03

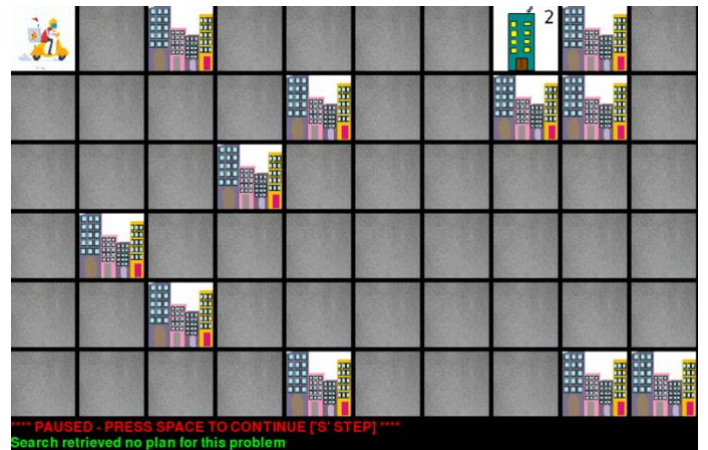
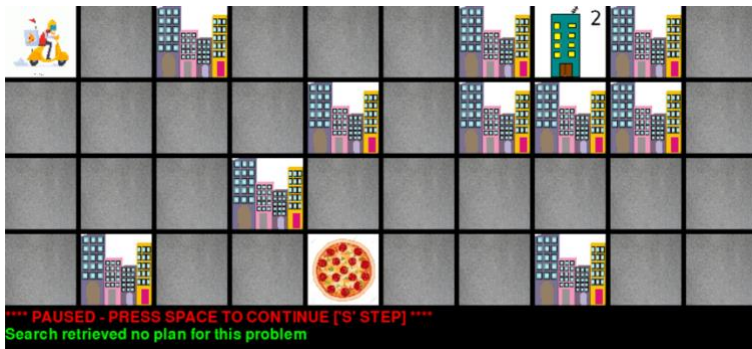
Testing for another case with different map size using A\* algorithm.



By testing the behavior on this map, we have checked it works for different map sizes taking into consideration the new limits. The printState function, the number of pizzas picked, as well as the decrease on number of pizzas to be delivered to the customer are still working as expected; from which, we can extrapolate that our implementation is good.

## TEST-04

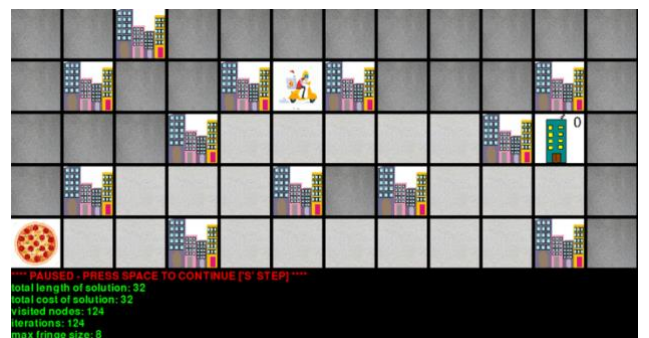
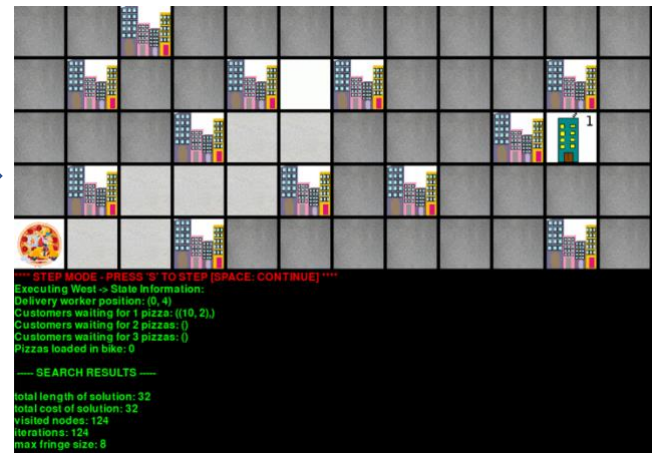
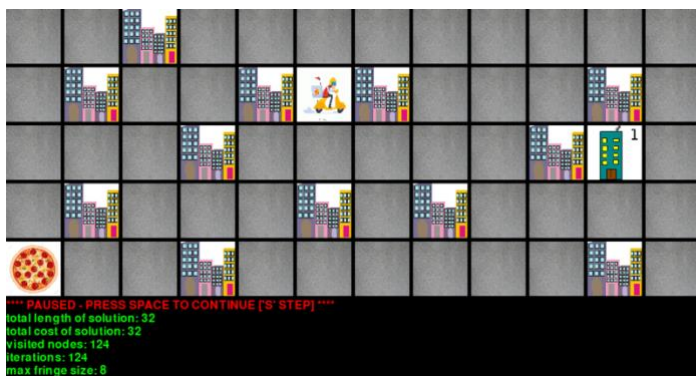
Testing for maps without solution.



As we can see, it behaves as expected since no solution is retrieved for impossible cases.

## TEST-05

Testing different deliverer initial position, changing manually from the config file the information.





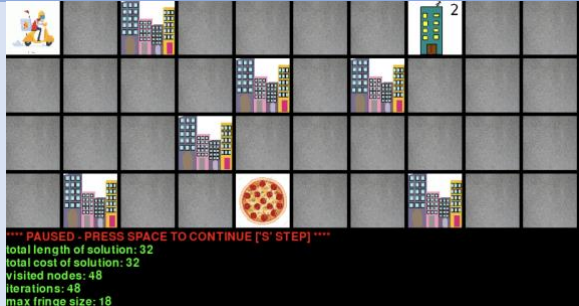
As we can see, it behaves as expected, going for the pizza, then to the client location and then back to its starting position.




## 2.4 Comparisons

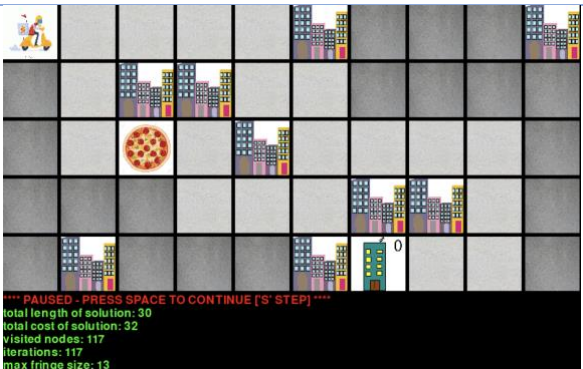

After some basic tests were done we proceeded with the comparison among the different provided algorithms on our code. To change the algorithm, we simply had to replace the variable ALGORITHM declared at the beginning of our code. Other comparisons are also made on the advance problem section.

### First comparison

| Algorithm     | Obtained result   | Comment  |
|---------------|---|--|
| A*            |    | As it was expected this approach is both the one that best performs in number of visited nodes and obtains the minimum length of solution. However, its max fringe size is not the smallest one which means it has may have used more memory than other algorithms such as breadth first search. However, breadth first search has more visited nodes. |
| Breadth first |   | This algorithm needs to visit more nodes to come up with the optimal solution, however it also achieves the minimum length solution, as it should.   |
| Depth first   |  | Depth first search algorithm comes close to having the least number of visited nodes, but it does not find the optimal solution but another one with a greater length. This is also expected given the behavior of the algorithm which does not guarantee the optimal solution.  |

### Second comparison

| Algorithm | Obtained result   | Comment   |
|-----------|---|---|
| A*        |  | As in the previous example, this approach is both the one that best performs in number of visited nodes and obtains, as expected, the minimum length of solution. Again the max fringe size is not the smallest which means more memory was used. |

|                             |   |  |
|-----------------------------|---|--|
| <p><b>Breadth first</b></p> |  | <p>Again, this algorithm needs to visit more nodes to come up with the optimal solution, however, it also achieves the minimum length solution as it should.</p>   |
| <p><b>Depth first</b></p>   |  | <p>Similar to the previous comparison, depth first search algorithm comes close to having the least number of visited nodes, but it does not find the optimal solution but another one with a greater length. This is also expected given the behavior of the algorithm.</p> |

### 3. Part II – Advanced Problem

After successfully implementing the basic functionality, we aimed for the advanced problem functionality. However, some parts of this functionality were already thought at the beginning such as the possibility of having multiple clients with different number of orders or different restaurant to pick pizzas from. This early thought on the advanced functionality was done to avoid changing a lot of the implemented functions. The modified functions are detailed below.

#### 3.1 Updated cost(self)

This function will return the cost of a given action. If we are on a hill cell, it will return 5. Otherwise if we are on an off-road cell, it will return 2. If we are not in any of those types of cell, we will return 1. There is no need to check whether or not we are on a sea cell as it is checked in the actions function, where we make sure there is no scenario in which we will allow movement towards such cells.

#### 3.2 Update heuristic(self, state)

For the advance problem, a new heuristic was developed in which we will return the Manhattan distance to the start cell from the current position plus the number of remaining customers. This heuristic is valid for multiple customers with different number of orders and multiple restaurants.

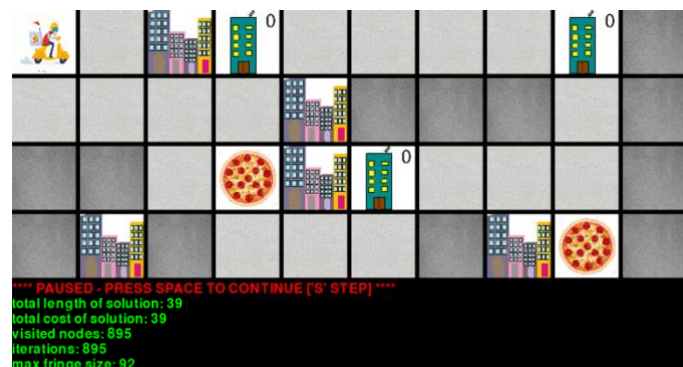
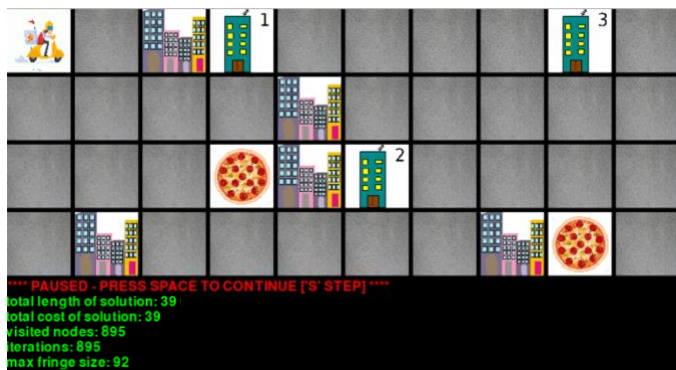
We tried many other heuristics which considered pizzas and position of the clients among others, but they were much more complex and not admissible in some situations. However, generally, the number of visited nodes was lower.

### 3.3 Test for multiple clients and restaurants

After the code was implemented and we believed it worked as expected, it was time to test its behavior on different environments through different tests. If tests were successful they will appear on **green** otherwise on **red**.

#### TEST-01

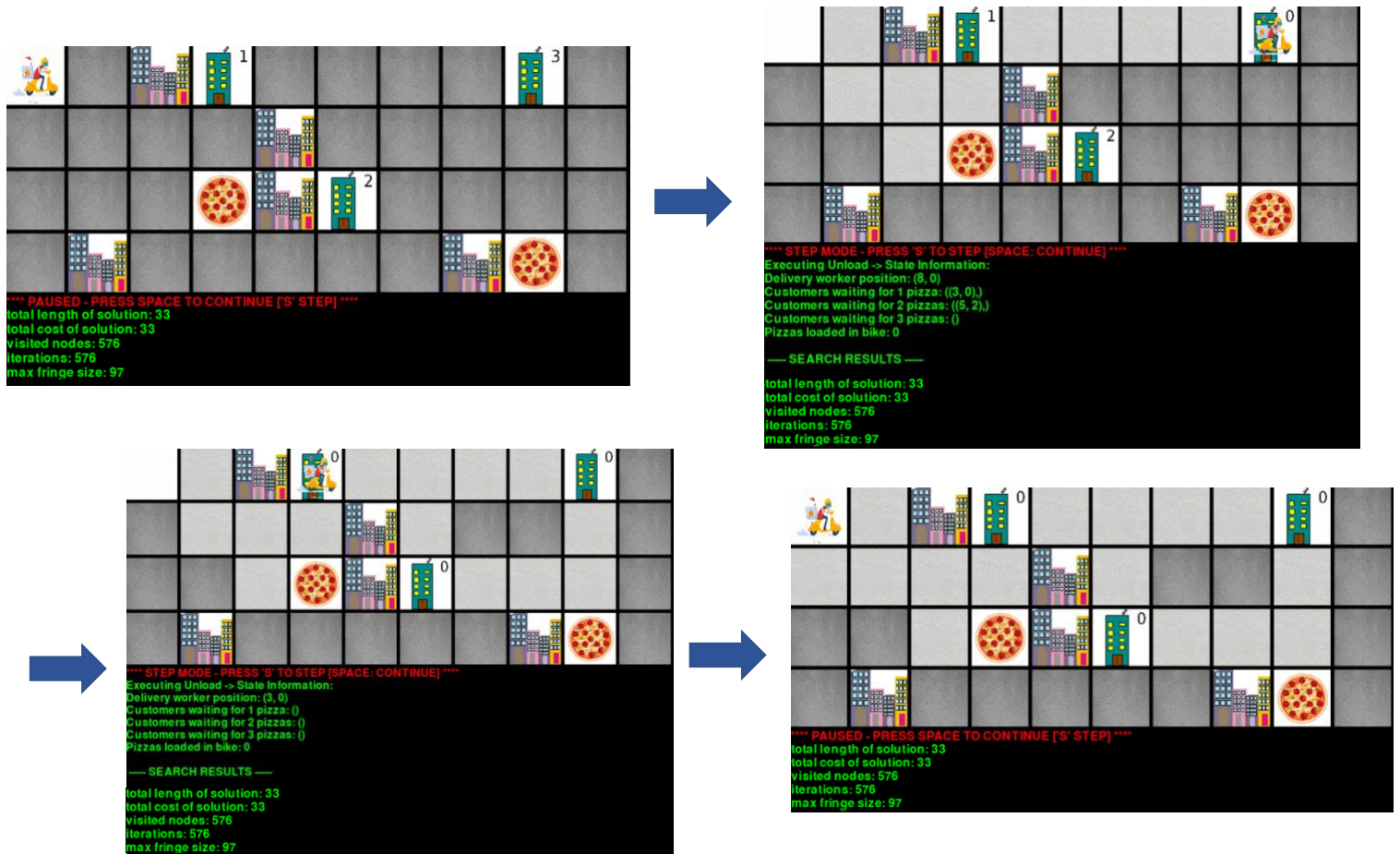
Testing a case with multiple clients and pizza restaurants.



As we can see it behaves as expected, going for more pizzas when it needs them and successfully delivering all orders.

## TEST-02

Testing same case as before but setting the `MAXBAGS` to three so the deliverer can carry more pizzas

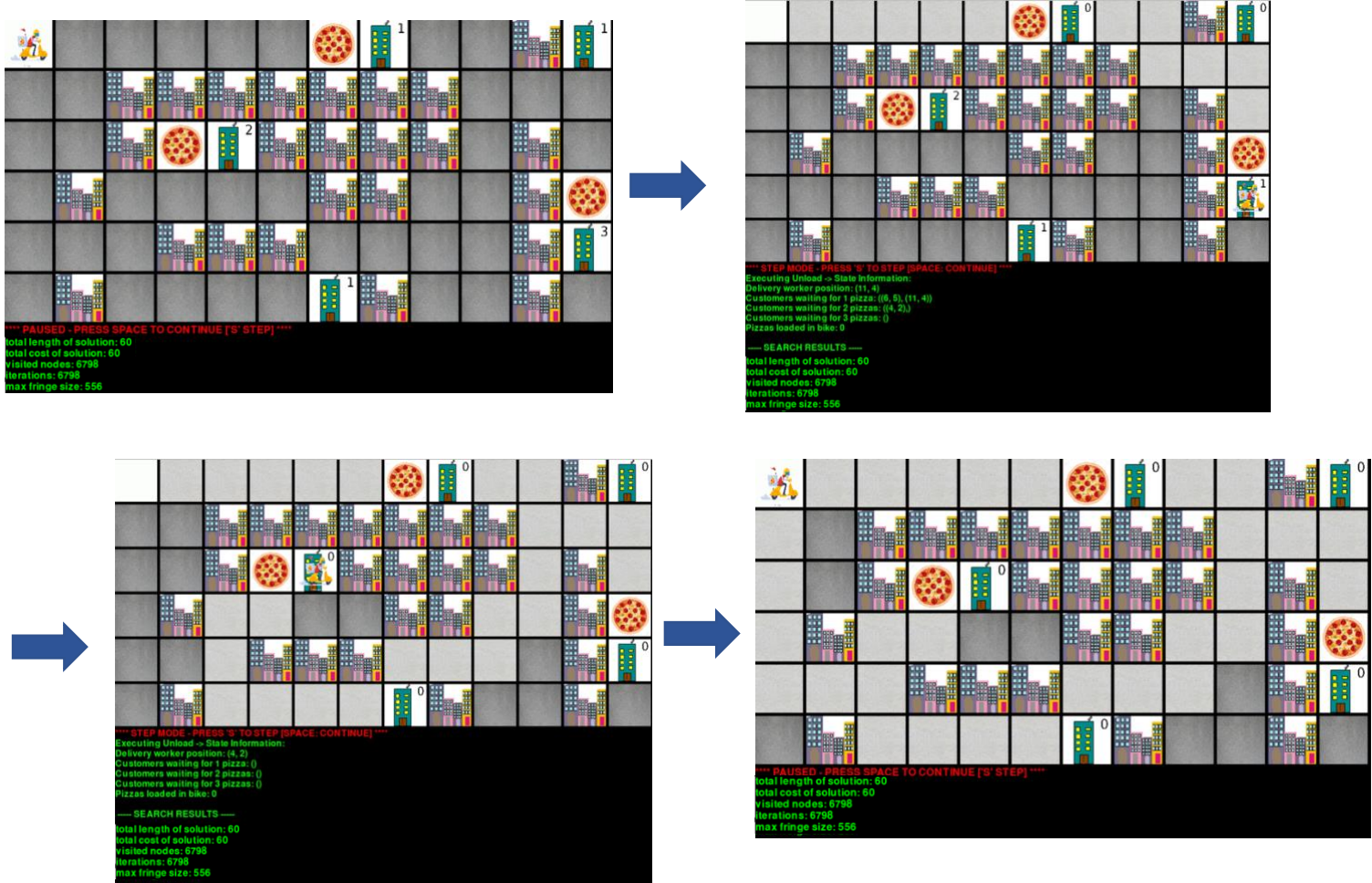


As we can see it behaves as expected, going for more pizzas we it needs it and successfully delivering all orders, this time however it only needs to stop for pizza two times



### TEST-03

Testing a case with multiple clients and pizza restaurants but much greater.



As we can see it behaves as expected, going for more pizzas we need it and successfully delivering all orders.

### 3.4 Multiple types of terrains and costs

We allowed the user to input three new different types of terrain. To configure, the user will have to change the map.txt file by putting H where it wants a hill, S sea and O for off road tiles. As expected, all previous tiles can still be used.

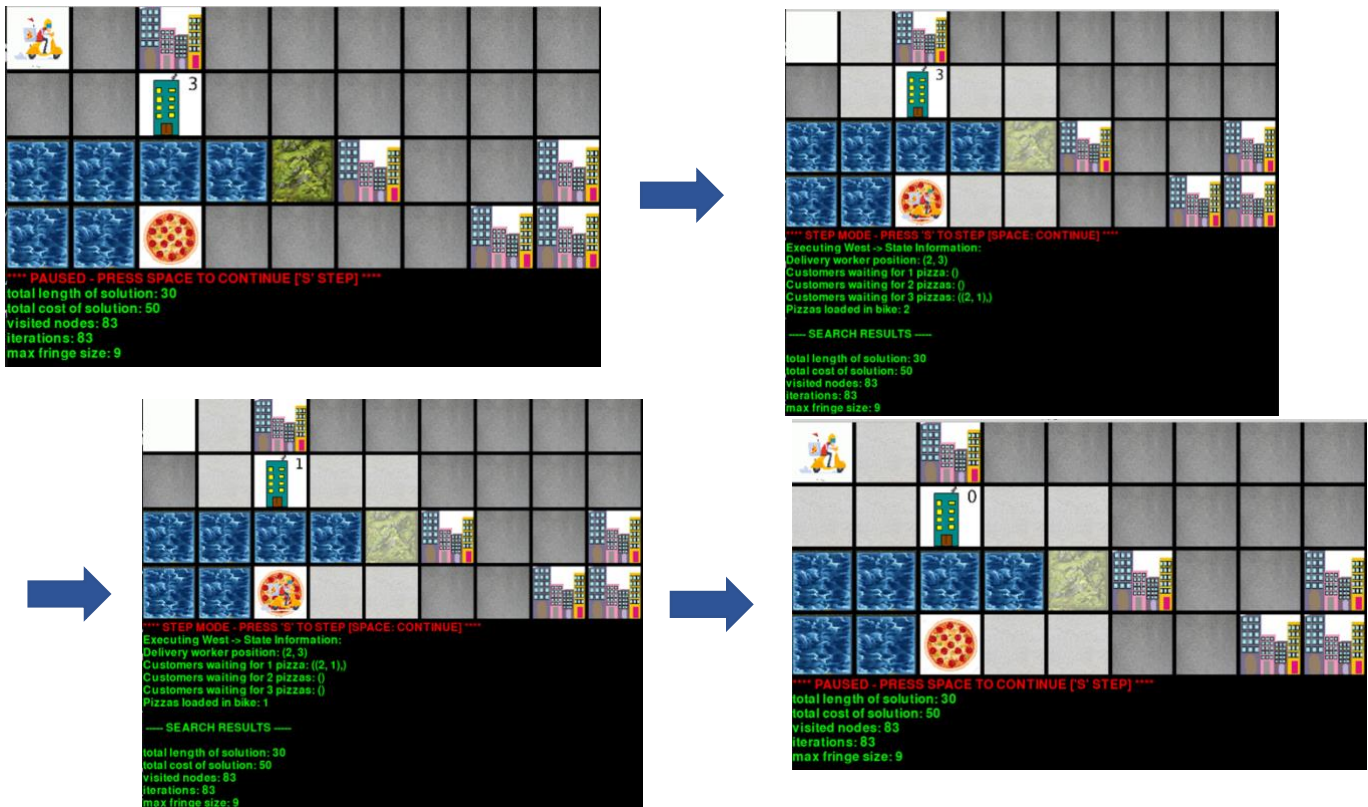
Whenever there is a hill tile the cost will be five as it will take much petrol to move the bike up. Off road is not really an issue for the bike as it is lightweight, however the boss will get angry if the bike is full of dust and therefore the cost of going through such tiles is two. Lastly, we will avoid completely going through the sea as the motorbike will stop working and the deliverer will be fired. All three new types of tiles have been added into the config class in which, after street, they have been declared, so we can recognize the user input. The first two of them (hill and offroad) will be added to our cost function in which we will check whether we are in one of them or not returning the pertinent value if we are (5 or 2). Regarding the sea tiles we will look for them on the action function, in which we will verify not only that the tile is within range and is not a building but also that it is not a sea tile.

## TESTING:

After the code was implemented and we believed it worked as expected, it was time to test its behavior on different environments through different tests. If tests were successful they will appear on **green** otherwise on **red**.

### TEST-01

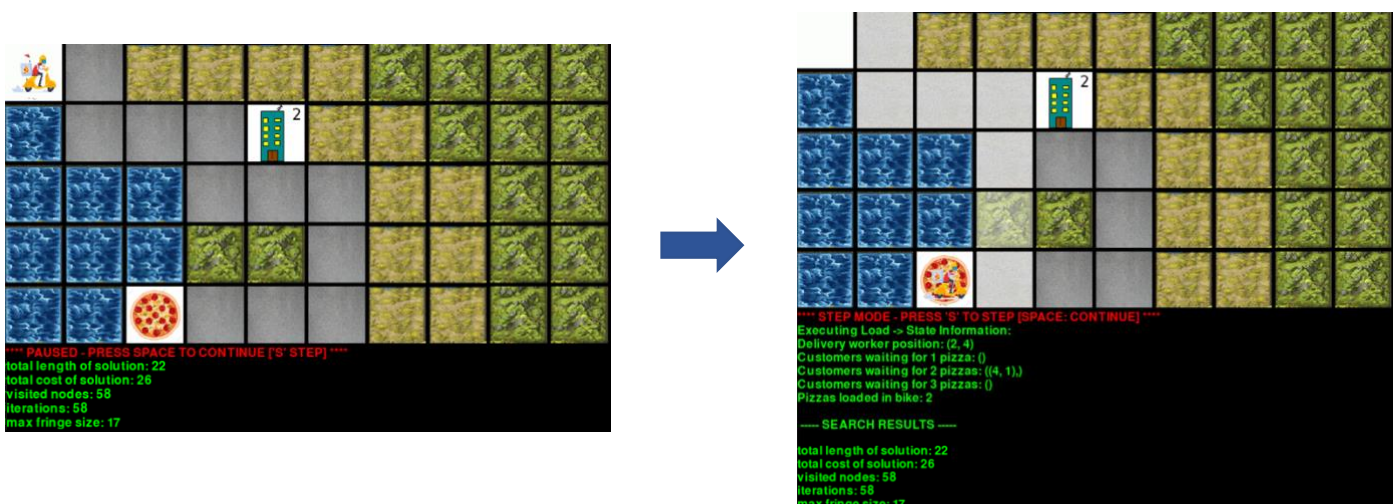
Testing that sea is avoided at any cost but hill path is taken as it's cheaper than surrounding.



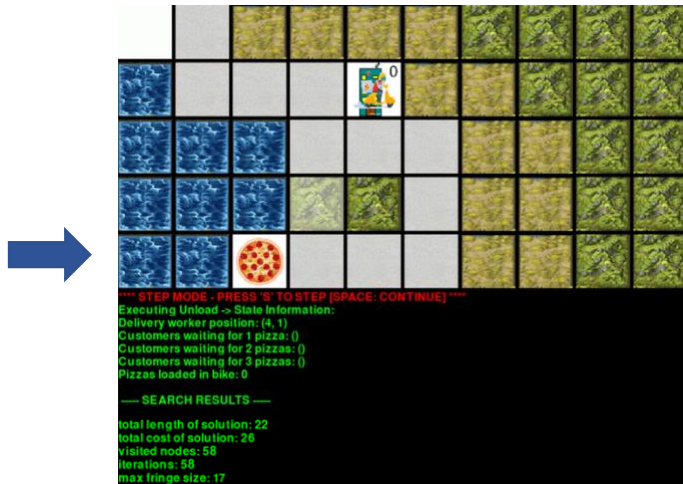
As we can see it behaves as expected since it avoids the sea at all costs. Nevertheless, it goes through the hill as it would be more costly to surround it.

### TEST-02

Testing that it avoids the hill if another path is cheaper and it again does not go through sea tiles



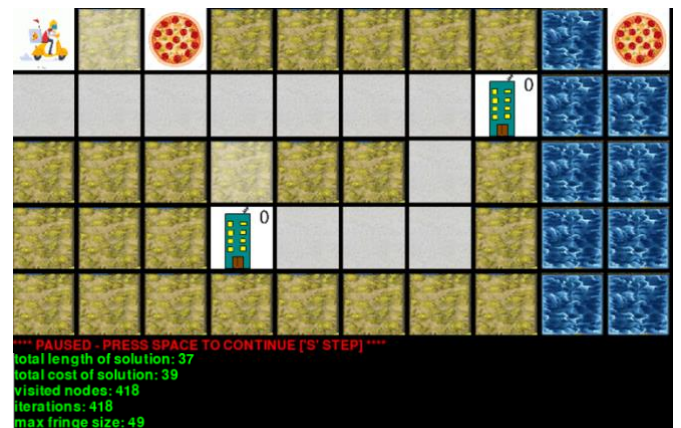
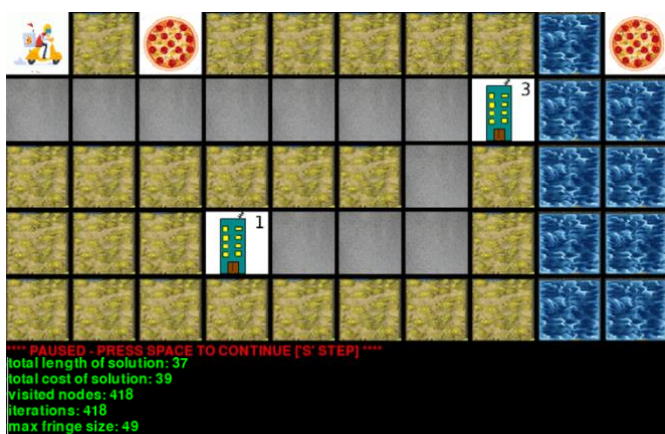




As we can see it behaves as expected since it avoids the sea at all costs and it does not go through the hill for picking the pizza as it would be more costly than surrounding it. However, to deliver to the customer, as it is cheaper to surround it, it does so.

### TEST-03



Testing that it avoids the off-roading when road is cheaper but takes it if necessary.



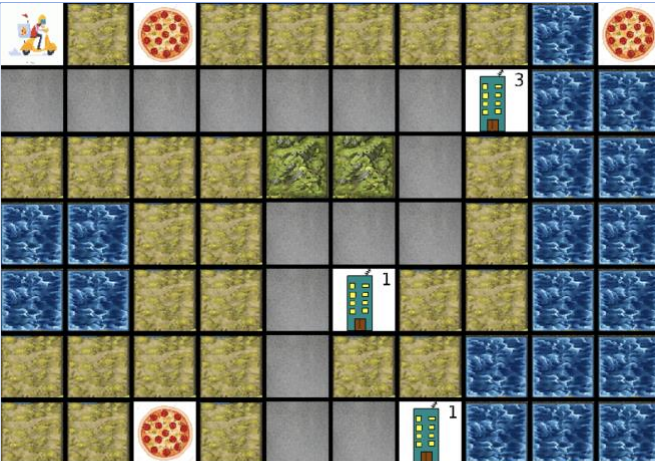
It behaves as expected taking off-road when necessary.


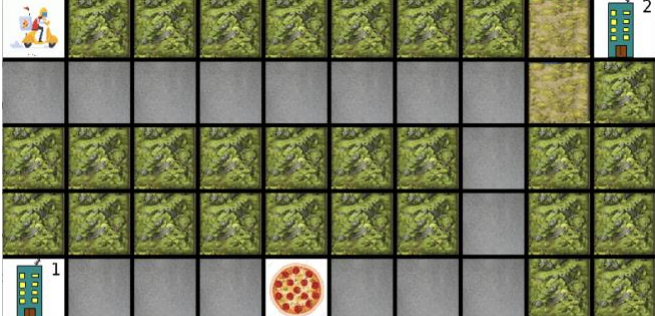
### 3.5 Comparison

After some basic tests where done, we proceed with comparisons among the different provided algorithms on our code. To change the algorithm, we simply had to replace the variable `ALGORITHM` declared at the beginning of our class. As sea is check along with building blocks, no matter the algorithm used it will avoid the cells (as well as the buildings).

| Algorithm     | Obtained result  | Comment   |
|---------------|--|---|
| A*            |  <p>**** PAUSED - PRESS SPACE TO CONTINUE ['S' STEP] ****<br/> total length of solution: 51<br/> total cost of solution: 56<br/> visited nodes: 1545<br/> iterations: 1545<br/> max fringe size: 114</p>  | As it was expected, this approach is both the one that best performs in number of visited nodes and obtains the minimum length of solution. However, it is the one using more memory (it has the maximum fringe size and visited nodes).                    |
| Breadth first |  <p>**** PAUSED - PRESS SPACE TO CONTINUE ['S' STEP] ****<br/> total length of solution: 51<br/> total cost of solution: 74<br/> visited nodes: 1300<br/> iterations: 1300<br/> max fringe size: 67</p> | This algorithm needs to visit less nodes to come up with a solution. Nevertheless, as it doesn't take into account cost, it is much worse regarding that value. However, it achieves a better number of visited nodes, as well as the smallest fringe size. |



|             |  |  |
|-------------|--|--|
| Depth first |  <p>**** PAUSED - PRESS SPACE TO CONTINUE ['S' STEP] ****</p> <p>total length of solution: 111<br/>total cost of solution: 186<br/>visited nodes: 157<br/>iterations: 157<br/>max fringe size: 87</p> | <p>Depth first search algorithm comes close to having the least number of visited nodes, but it does not find the optimal solution but another one with a greater length and a much greater cost. This is also expected given the behavior of the algorithm.</p> |
|-------------|--|--|

| Algorithm     | Obtained result   | Comment   |
|---------------|---|---|
| A*            |  <p>**** PAUSED - PRESS SPACE TO CONTINUE ['S' STEP] ****</p> <p>total length of solution: 46<br/>total cost of solution: 50<br/>visited nodes: 276<br/>iterations: 276<br/>max fringe size: 31</p>   | <p>As it was expected this approach is both the one that best performs in number of visited nodes and obtains as expected the minimum length of solution. Similar to the previous comparison it is not the one using less memory as its max fringe value is greater than breadth first search one as well as the visited nodes.</p> |
| Breadth first |  <p>**** PAUSED - PRESS SPACE TO CONTINUE ['S' STEP] ****</p> <p>total length of solution: 38<br/>total cost of solution: 116<br/>visited nodes: 345<br/>iterations: 345<br/>max fringe size: 19</p> | <p>This algorithm needs to visit less nodes to come up with a solution. Nevertheless, as it doesn't take into account cost, it is much worse. However, it achieves a smaller number of visited nodes, as well as the smallest fringe size.</p>  |

|                           |   |   |
|---------------------------|---|---|
| <p><b>Depth first</b></p> |  | <p>Depth first search algorithm comes close to having the least number of visited nodes, but it does not find the optimal solution but another one with a greater length. This is also expected given the behavior of the algorithm</p> |
|---------------------------|---|---|

## 4. Conclusion

This lab assignment has allowed us to familiarize with python, search algorithms, and code applied to artificial intelligent problems. After doing it we believe to have a better understanding of how python works, as well as how the search algorithms behave given the state. We believe the hardest part of the practice was not to code itself but understanding how python behave as we have never used it before. Moreover, understanding the given code and how the algorithm would behave given a state was also complicated. We believed however to have developed a pretty efficient and clean solution with few node expansions for the given problem, behaving as expected in all cases for all the given algorithms.

We have used as support materials the provided pdf as well as many online sources for our understanding of python along with the provided code, which was well commented and eased the development of the practice.

## 5. Personal comments

We found out this lab to be fun and truly useful for our degree once we figured out how install all the necessary components. Also, with the university account we all have free access to truly powerful integrated development environments such as PyCharm or some others (even some free open sourced ones) which took us some time to find and learn how to use but were extremely helpful for coding and debugging. We believe spending some time showing any of them and how to configure them to run the given code would be really useful as otherwise the student has to spend a lot of time just to be able to start the practice. Nevertheless, we enjoyed this practice and we believe learning python and search algorithms in code is really useful and will turn handy in the near future, something that is sometimes difficult to notice in some other practices we have to do.