<u>Assignment 2</u>

Louka CHAPUS

1. Part 1 – Grid with obstacles

1.1 Task 1:

For this task we were asked to implement the A* algorithm to find the path between two points. So first we need a function to calculate the distance heuristic between the start and the end, we use the Manhattan definition to calculate the distance.

```
def heuristic_h(self, x: int, y: int):
    h = abs(x - self.end_goal_pos[0]) + abs(y - self.end_goal_pos[1])   # Heuristic (Manhattan) given the final goal position
    return h
```

We just need a position as an input and the function do the rest and it output the distance. Then we have the actual implementation of the A* algorithm, we use the module named heapq to keep a list ordered in ascending order each time we add an element to the list.

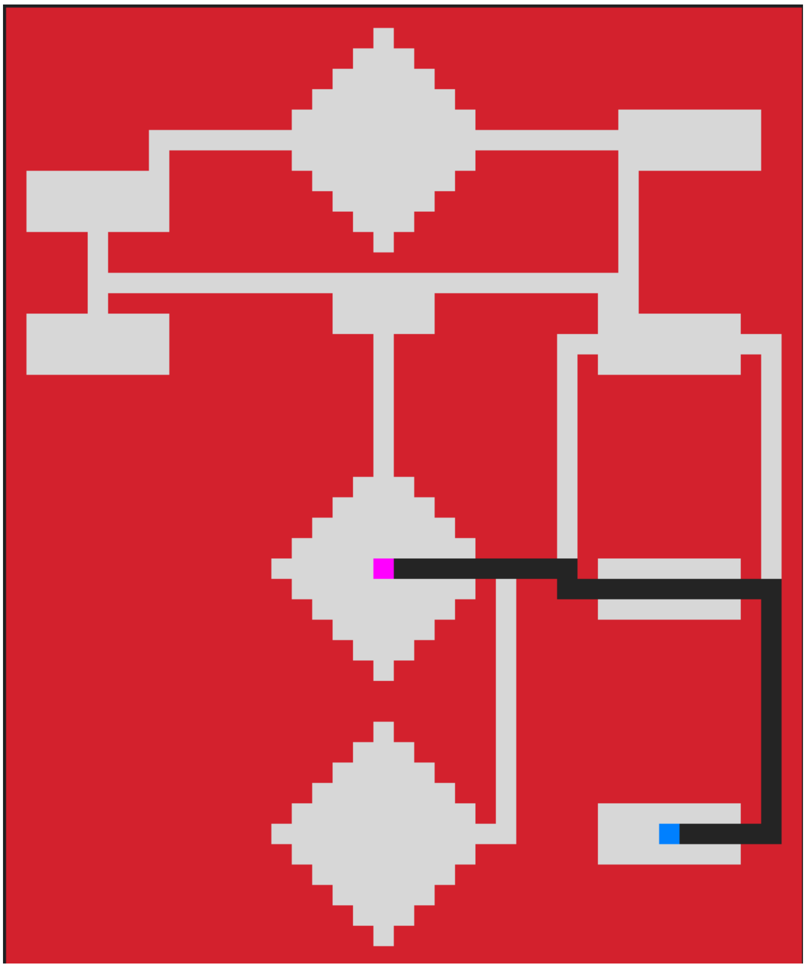
```
def path_finder_without_cost(self):
    open_set = []
    closed_set = set()
    node_with_parent = []

    start_node = (self.start_pos[0], self.start_pos[1])
    start_h = self.heuristic_h(start_node[0] ,start_node[1])    # Estimation of the remaning cost
    heapq.heappush(open_set, (start_h, start_node))             # Add h and the starting node to the open set

    node_with_parent.append((start_node, None))                 # Used to store all the dependencies between each point

    while open_set:

        current_h, current_node = heapq.heappop(open_set)       # Remove the fisrt node in the open set

        if (current_node[0] == self.end_goal_pos[0] and current_node[1] == self.end_goal_pos[1]):   # If we are at the end postion
            path = []
            current_node = node_with_parent[-1][1]                                                  # Take the last node before the end

            while current_node:                                                                     # While there is a node (the start have None for parent)
                for k in range(len(node_with_parent)):                                              # Test all the node
                    if (node_with_parent[k][1] == None):                                            # Because the start have a None parent
                        if (current_node[0] == start_node[0] and current_node[1] == start_node[1]): # If we are at the start
                            path.append(start_node)                                                 # Add the start at the path
                            return path[::-1]                                                       # Return the reverse the path (i.e. the path from the start to the end)

                    if (node_with_parent[k][0][0] == current_node[0] and node_with_parent[k][0][1] == current_node[1]):
                        path.append(current_node)                                                   # Add the current node at the path
                        current_node = node_with_parent[k][1]                                       # Switch between the parent and the current node


        closed_set.add(current_node)                            # Add the current node to the closed set so we don't try it again

        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:       # Try every neighbor (right, left, top, bottom)
            new_x, new_y = current_node[0] + dx, current_node[1] + dy


            if (0 <= new_x < len(self.int_map) and 0 <= new_y < len(self.int_map[0]) and self.int_map[new_x][new_y] != -1): # Check if the new coordinates are in a valid range or if this is a wall (i.e. egal to -1)
                neighbor = (new_x, new_y)                       # update the neighbor with the new coordinates

            if neighbor in closed_set:                          # If the neighbor is in the closed set, we already test it
                continue                                        # Skip to the next iteration

            if not any(node[1] == neighbor for node in open_set):
                neighbor_h = self.heuristic_h(neighbor[0], neighbor[1])     # Estimation of the remaning cost
                heapq.heappush(open_set, (neighbor_h, neighbor))            # Add the neighbor with his cost in the open set
                node_with_parent.append((neighbor, current_node))           # Add the neighbor with his parent in the list node_with_parent

    return None
```

So first we initialize all the elements we need, the open set is used to discover all the tiles and when a tile is explored then we put it into the closed set and the list node_with_parent is here to save all node with their parent, it will be used to print the path. Then the first node in the open set is always the closer to the end point, so we always try this one first and we add all this neighbour to the open set (if they are in a valid position and if they aren't in the closed set). And we keep going until we reach the end point. When we are at the end point then we need to output the path between the star and the end. To do that, we have all the nodes with their parents, so we just need to find the parent of each current node and we have the path. But we need to reverse it because it's the path from the end to the start. Finally, we need to print the result on the map and show the map.

```python
def print_path_without_cost(self):
    path = self.path_finder_without_cost()
    if path:
        for k in range(1,len(path)):
            self.replace_map_values(path[k],4,(self.end_goal_pos))
        self.show_map()
    return None
```

Here we execute the previous function to have the path (if no path is finding the function return none). For each position in the path, we set the value of the map at 4 (i.e. black) and we show the map. So, we just must call this function to show the map with the path printed on it.

```python
map_obj_task_1 = Map_Obj(task=1)
map_obj_task_1.print_path_without_cost()
```

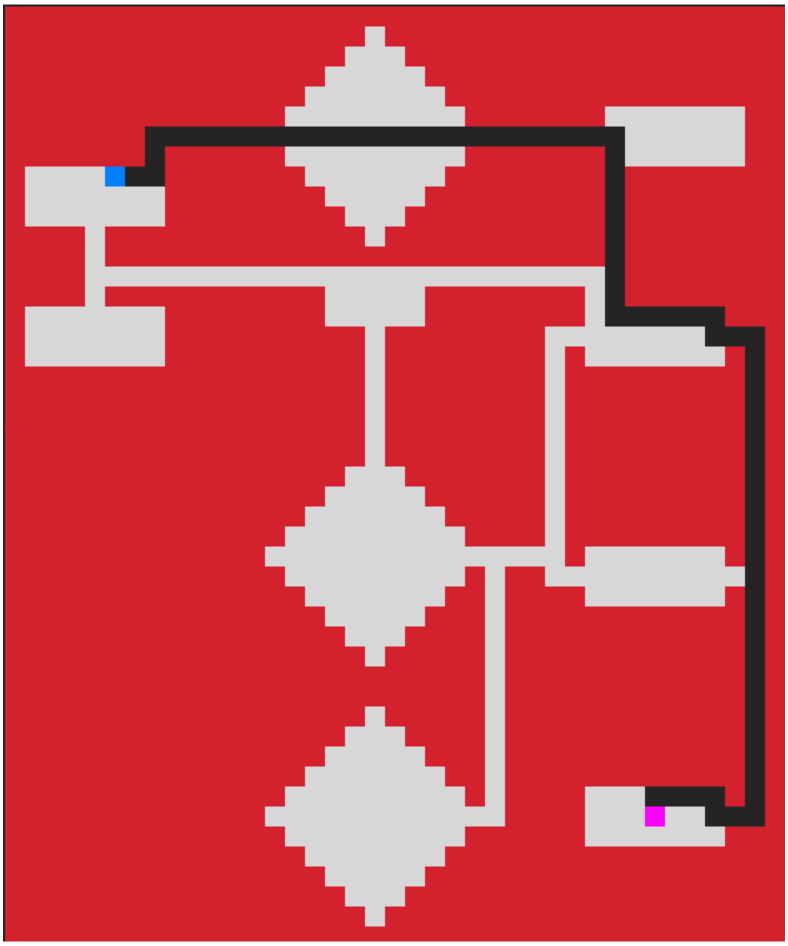


This is the result of the execution for the first task. It finds the shortest path between the start and the end.

1.2 Task 2:

We used the exact same code than previously, the only thing that change it's when we create the object of the class Map_Obj, we set the task to 2.

```python
map_obj_task_2 = Map_Obj(task=2)
map_obj_task_2.print_path_without_cost()
```



This is the result of the second execution of the A* algorithm, we can think that is not the shortest path. But it is like that because we use the Manhattan function to calculate the distance between a point and the end. If we were using the Euclidean distance, we will have a more diagonal path.

2. Part 2 – Grid with different costs

In this part we are now including the cost of each cell. Our goal is to find the shortest path with the less total cost.

2.1 Task 3:

We use the same function to calculate the distance heuristic than before with the Manhattan definition. So, now we need to add an extra information to the open set, we define f as the sum of the remaining cost and the real cost it takes to go here. At the beginning we introduce all the variables we are using.

```python
open_set = []
closed_set = set()
node_with_parent = []

start_node = (self.start_pos[0], self.start_pos[1])
start_g = 0                                                    # Real cost of the starting point
start_h = self.heuristic_h(start_node[0] ,start_node[1])       # Estimation of the real cost
start_f = start_h + start_g                                    # Caculate the total cost
heapq.heappush(open_set, (start_f, start_g, start_node))       # Add f, g and the starting node

node_with_parent.append((start_node, None))                    # Used to store all the dependencies between each point
```

The starting point have no cost, that's why g is 0 and it have no parent (i.e. none in node_with_parent). Then we have the exact same function for printing the path. And we have then end of the function were things got a little more complicated.

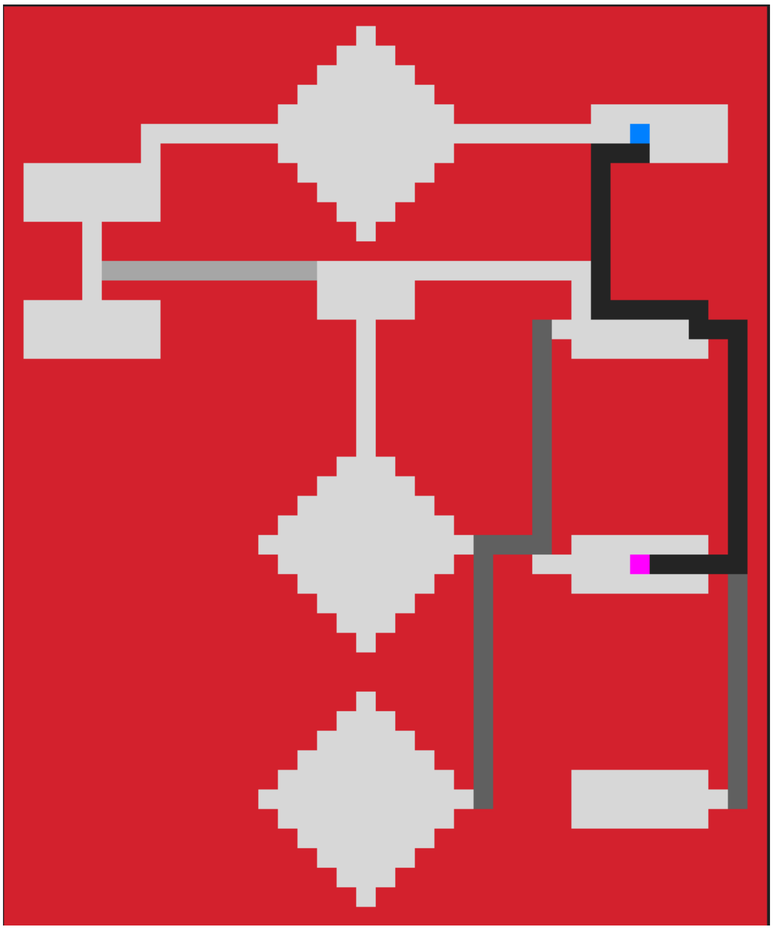
```python
for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
    new_x, new_y = current_node[0] + dx, current_node[1] + dy


    if (0 <= new_x < len(self.int_map) and 0 <= new_y < len(self.int_map[0]) and self.int_map[new_x][new_y] != -1):
        neighbor = (new_x, new_y)

    if neighbor in closed_set:
        continue

    neighbor_g = current_g + self.get_cell_value(neighbor)      # Current cost plus travel cost
    neighbor_h = self.heuristic_h(neighbor[0], neighbor[1])     # Estimation of the remaining cost
    neighbor_f = neighbor_g + neighbor_h                        # Total cost

    if not any(node[2] == neighbor for node in open_set) or neighbor_g < current_g:
        heapq.heappush(open_set, (neighbor_f, neighbor_g, neighbor))
        node_with_parent.append((neighbor, current_node))
```

For each valid point we need to determine the remaining cost (i.e. h) and we need to update the travel cost (i.e. g), then we push it to the open set. The open set is sorted by the total cost for each point, so we always test the point with the shortest distance in first. Finally, we just need to make almost the same function than before to show the map with the path printed on it.

```python
def print_path_with_cost(self):
    path = self.path_finder_with_cost()
    if path:
        for k in range(1,len(path)):
            self.replace_map_values(path[k],4,(self.end_goal_pos))
        self.show_map()
    return None
```

We just have changed the second line, we call the function we just have define. And then we just need to call this function to show the map with the path printed on it in black.

```python
map_obj_task_3 = Map_Obj(task=3)
map_obj_task_3.print_path_with_cost()
```

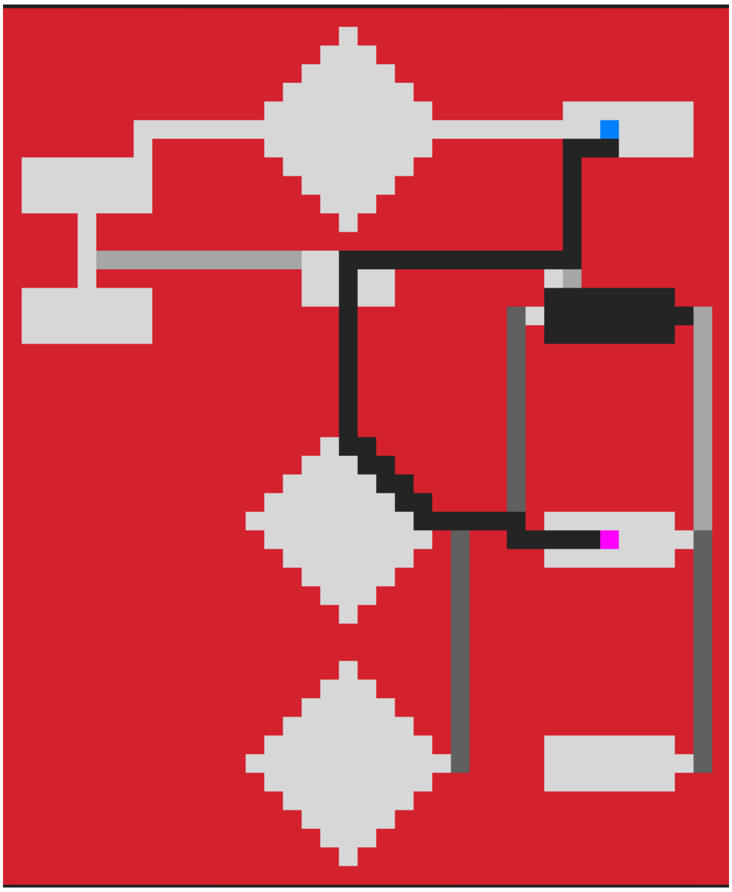


This is the result of the execution, we see that the shortest path without considering the cost is by left. But we consider the cost of each cell in this part, so the shortest path is now by right and that's what the A* algorithm find so it seems to be working.

2.2 Task 4:

For this last task, we use the function define before. We just need to change the number of the task when we create the object.

```
map_obj_task_4 = Map_Obj(task=4)
map_obj_task_4.print_path_with_cost()
```



This is what we got after the execution. Again, the shortest path without considering the cost is all the way up. But with considering the cost we end up with the solution find by the algorithm, the path is printed in black.