Introduction to Artificial Intelligence
Lecture 5: Exercises on search algorithms

Exercise 1: Edit the A* algorithm code to return the shortest path found when reaching the goal node.

Solution:

To return the shortest path when the goal node is reached, the A* algorithm must keep track of how each node was reached. This is done by storing the parent of each node whenever a better path to that node is found. Once the goal node is reached, the shortest path can be reconstructed by backtracking from the goal node to the start node using these parent values, and then reversing the resulting sequence of nodes.

The following code implements this idea through:

- editing the original A* function to return a dictionary containing the parent of each node.
- defining a new function “reconstruct_path” that calculates the path given the parent dictionary and the end node.

```python
from queue import PriorityQueue

def a_star_search(graph, start_node, end_node, heuristic):

    solution = []

    frontier = PriorityQueue()
    visited = []
    # Defining the parent of each node
    parent = {}
    # cost from start to each node
    g_cost = {start_node: 0}

    # f(n) = g(n) + h(n)
    f_start = g_cost[start_node] + heuristic[start_node]
    frontier.put((f_start, start_node))

    # Setting the parent of the start node
    parent[start_node] = None

    while not frontier.empty():

        _, selected_node = frontier.get()

        if selected_node in visited:
            continue

        visited.append(selected_node)
        solution.append(selected_node)

        if selected_node == end_node:
            break

        for neighbour in graph[selected_node]:
            if neighbour in visited:
                continue

            # cost from start to neighbour through selected_node
            tentative_g = g_cost[selected_node] + 1    # each step costs 1

            # If neighbour has no g_cost yet, give it a very large value
            if neighbour not in g_cost:
                old_cost = 999999
            else:
                old_cost = g_cost[neighbour]

            # Only update if this path is better
            if tentative_g < old_cost:
                g_cost[neighbour] = tentative_g
                f_value = tentative_g + heuristic[neighbour]
                frontier.put((f_value, neighbour))
                parent[neighbour] = selected_node    #updating the parent of the node

    return solution, parent
```

```
def reconstruct_path(parent,current):
    path = [current]
    while parent[current] is not None:
        path.append(parent[current])
        current = parent[current]

    path.reverse()
    print(path)
```

Applying the functions to an example graph:

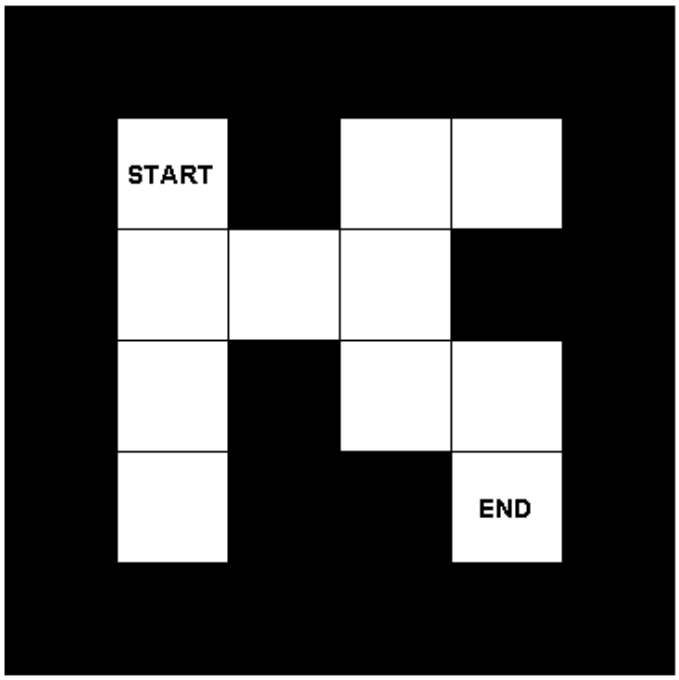
```
# Example graph (state space)
state_space = {
    "A": ["B"],
    "B": ["C"],
    "C": ["D", "G"],
    "D": ["E"],
    "E": ["F"],
    "F": [],
    "G": ["H"],
    "H": ["K", "I"],
    "K": ["L"],
    "L": ["M"],
    "M": ["N"],
    "N": [],
    "I": ["J"],
    "J": [],
}

heuristic = {
    "A": 8,
    "B": 7,
    "C": 6,
    "D": 5,
    "E": 4,
    "F": 5,
    "G": 5,
    "H": 4,
    "K": 3,
    "L": 2,
    "M": 1,
    "N": 0,
    "I": 5,
    "J": 4,
}
start_state = "A"
goal_state = "N"

s,p =a_star_search(state_space, start_state, goal_state,heuristic)
print(s)
reconstruct_path(p,s[-1])
```

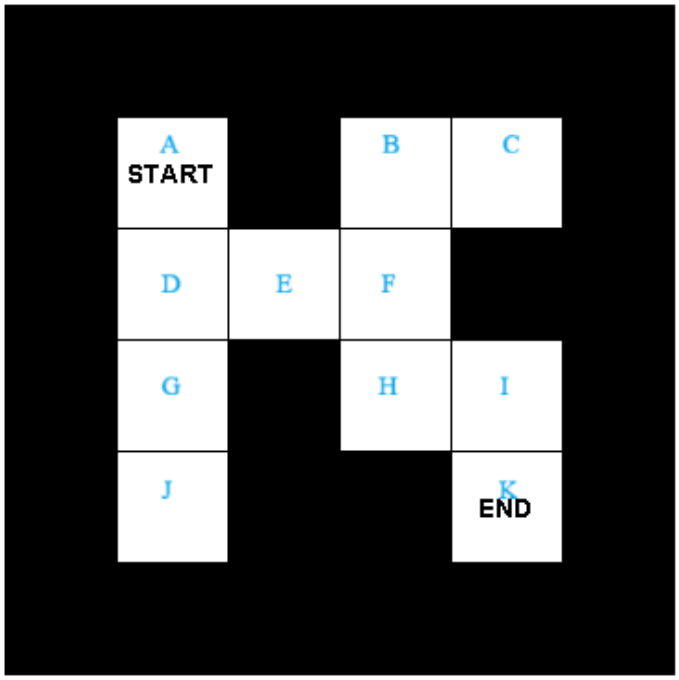
Exercise 2: For the following grid-based maze:

- Construct a dictionary that represents the maze.
- Define a function that calculates the Manhattan Distance between two positions (x1, y1) and (x2, y2).
- Using the function you defined, construct a dictionary that assigns a heuristic value for each node.



Solution:

1- First, we define each free square as a node in the graph.



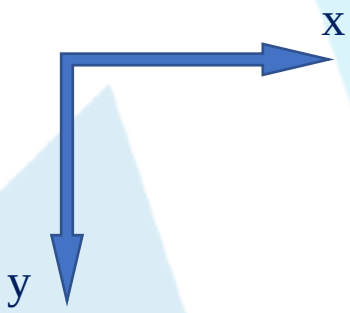
Based on the defined labels, the dictionary representing the maze is:

```
maze = {"A":["D"],
        "B":["C","F"],
        "C":["B"],
        "D":["A","E","G"],
        "E":["D","F"],
        "F":["B","E","H"],
        "G":["D","J"],
        "H":["F","I"],
        "I":["H","K"],
        "J":["G"],
        "K":["I"]
        }
```

2- The Manhattan Distance function takes four parameters x1, y1, x2, and y2, where: x1 and y1 are the coordinates of the first point, and x2 and y2 are the coordinates of the second point.
abs(x) is a built-in function in Python. It returns the absolute value of its input x.

```
def MD(x1,y1,x2,y2):
    return abs(x1-x2)+abs(y1-y2)
```

3- In order to calculate the heuristic (Manhattan Distance in this exercise), we need the coordinates of each node. Therefore, we will define a dictionary with node names as keys and their coordinates as values. To calculate the coordinates, we define the following arbitrary coordinate systems, with the origin at node A.



The dictionary representing the nodes coordinates is:

```
coordinates = {"A":[0,0],
               "B":[2,0],
               "C":[3,0],
               "D":[0,1],
```

```
"E":[1,1],
"F":[2,1],
"G":[0,2],
"H":[2,2],
"I":[3,2],
"J":[0,3],
"K":[3,3]
}
```

Now, that we have the node coordinates and the Manhattan Distance function, we can construct a dictionary that assigns the suitable heuristic value for each node.
The keys in the heuristic dictionary are the node names (same as in the coordinates dictionary).
The values in the heuristic dictionary are the Manhattan Distances from each node to the goal node K.
This code does the following:

- It creates an empty dictionary named heuristic.
- It loops over the keys in the coordinates dictionary, and in each iteration, it appends a new element into the heuristic dictionary.

```
heuristic = {}

for node in coordinates:
    heuristic[node] = MD(coordinates[node][0],coordinates[node][1],
                         coordinates["K"][0],coordinates["K"][1])

print(heuristic)
```

Code explanation:

`heuristic = {}`
Creates an empty dictionary where we will store the Manhattan Distance for each node.

`for node in coordinates:`
Loops through all the node names (A, B, C, …, K) in the coordinates dictionary.

Python loops through a dictionary by its keys by default, so in each iteration, the variable `node` will contain one key. The loop continues until all keys have been processed.

`coordinates[node][0]` and `coordinates[node][1]`
Each value in the `coordinates` dictionary is a list with two numbers:

`"A": [0, 0]  # [x, y]`

`coordinates["A"]` → `[0,0]`

When we write:

- `coordinates[node]` → the whole list `[x, y]`

- `coordinates[node][0]` → the first element in the list (the x-coordinate)
- `coordinates[node][1]` → the second element (the y-coordinate)

```
heuristic[node] = MD(coordinates[node][0],coordinates[node][1],
                     coordinates["K"][0],coordinates["K"][1])
```

For each node, we take its x and y coordinates and the coordinates of the goal node **K**, and pass them to the Manhattan Distance function `MD`.
The result is stored in `heuristic[node]`.

```
print(heuristic)
```

Prints the contents of the heuristic dictionary (node names and their heuristic values).

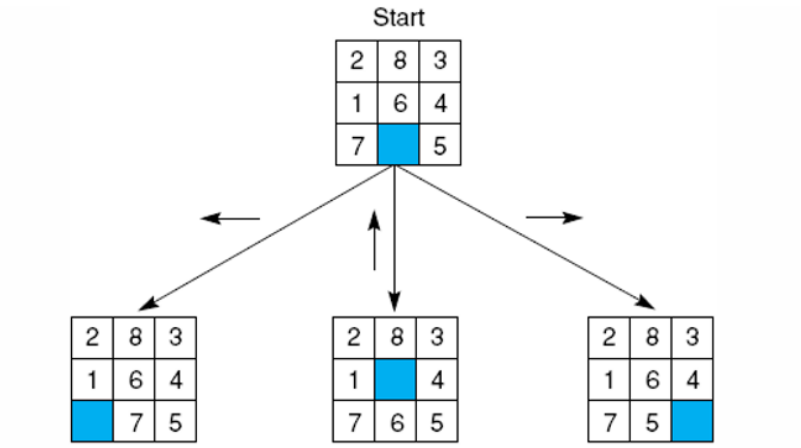
Code Output:

```
{'A': 6, 'B': 4, 'C': 3, 'D': 5, 'E': 4, 'F': 3, 'G': 4, 'H': 2,
 'I': 1, 'J': 3, 'K': 0}
```

Exercise 3: You have the 8-puzzle square shown below:

	2	**3**
1	**8**	**4**
7	**6**	**5**

At any moment, the gap can swap places with **one adjacent tile only,** for example:



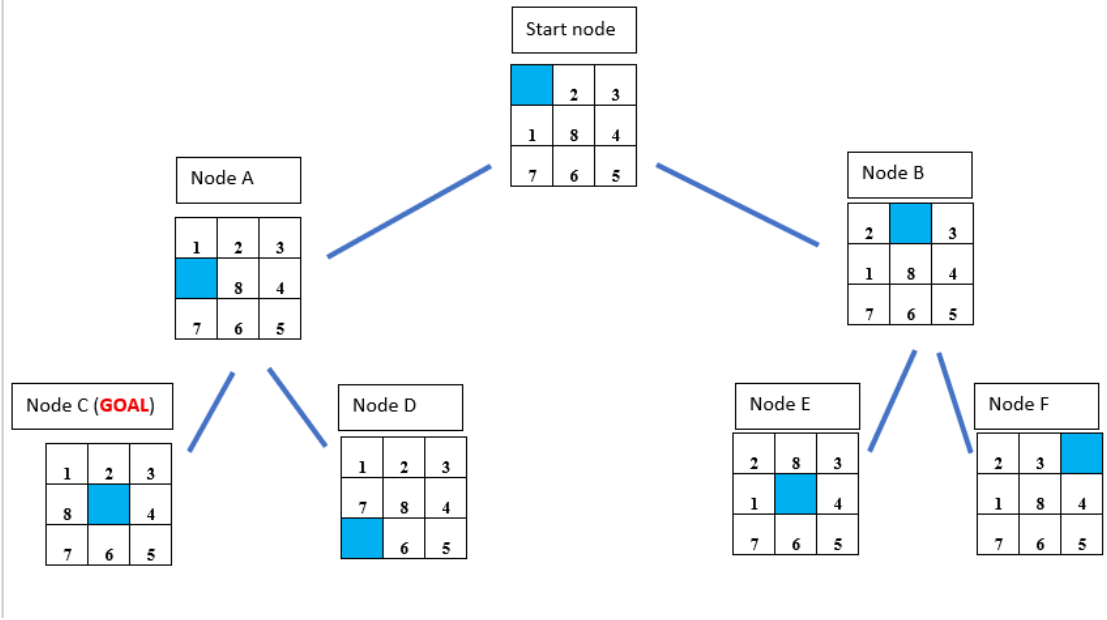
The goal is to reach the following configuration:

1	**2**	**3**
8		**4**
7	**6**	**5**

- Draw the graph starting from the initial state given till reaching the goal.
- Suggest a heuristic function to use in this problem.
- Calculate the heuristic value for each node in your drawn graph.

Solution:

1. First, we should draw the graph starting from the initial state given till reaching the goal.
   For every node, the neighbor nodes are the moves available starting from that node.
   We should pay attention to the fact that we should **not** put a child node that returns to the previous situation.



2. The heuristic is the estimated cost from some node to the goal node. When suggesting a heuristic function to any problem we should take a few points into consideration in order to suggest a meaningful heuristic.
   - The heuristic should be related to the problem we are solving.
   - The value of the heuristic should decrease when we are closer to the goal node (Closer to the solution of the problem).
   - The value of the heuristic should be zero at the goal node.

   For the previous problem, we are moving numbered squares to reach a specific configuration. There are several meaningful heuristic functions that may lead to a solution. For example (**number of squares out of their correct position**) or (**the sum of Manhattan distances of each square from its correct position**).

3. Taking the **first** suggested heuristic into consideration, the heuristic value of each node is as follow:

Node	Start	A	B	C (Goal)	D	E	F
heuristic	3	2	4	0	3	4	5

Note: we can count the gab square (the blue one) as an independent square, or we can ignore it. In the previous solution it was taken into consideration.

Exercise 4: For the built-in PriorityQueue class in Python, how can we make it behave as a max-priority queue instead of a min-priority queue?

Solution:

To make it behave like a max-priority queue, we can store each priority as its negative value.
When inserting an element, use the negative of its priority value. When removing an element, multiply the stored priority by -1 again to recover the original priority. Because the queue still selects the smallest stored value, the element with the largest original priority will be returned first.