# Introduction to artificial intelligence
# Write-Up : Lab 1

**last name : LE GUEN**
**surname : Eflamm**

**Abstract :**

During this first lab, we were asked to implement the A* algorithm in order to find the best possible path during an orientation race. The user must therefore provide a map of the environment including the level of elevation and the various terrain that can be found (lake, trail, forest, ect). The map therefore includes colors where for each color we can find the characteristics of the place in question. For each pixel(x,y) we can associate elevation and speed due to the type of environment.

**Input :**

1. A map of the orienteering (terrain.png).
2. The elevation corresponding to this map (mpp.txt).
3. The places to reach during the (path.txt).
4. The path of the output map (name_out.png).

On the output map it's written the **best route** and the **distance** if the user wants to take this path.

**Output :**

1. The output map displays on your screen.
2. The output map saved at the filename you gave in input.
3. The distance of the path is written on the output map.

## Implementation of the code :

**The class Node :**

First of all, I decided to create a class called Node. This class contains the global variables (speed, terrain, elevation, width, length, ect..) of our program, a constructor and somes functions. A node is defined by his position and by his parent(s). We can create a node by giving a position (X,Y) but this position has to fit with the shape of our map (395,500). Then the constructor finds the color associated with this pixel ,the corresponding speed and the elevation at this position. The nodes are as well defined by their weights (f,h and g) which are 0 by default. Each node is also defined by a unique key in order to work with dictionary in the algorithm A*. We can move in 8 directions to change the position of the current_node.

We also have overwritten some methods in order to compare Nodes **(__lt__,__gt__,__eq__).** These functions are used to work with our queue priority (heapq) in the second part of the code.

**The program A* :**

So, we can now create Nodes and we can access to different importants parameters to perform A* algorithm such as the value of the function f = cost+heuristic. My function A* takes two parameters to perform the algorithm : **start_note_position** and **end_node_position.** In a first time, we create two nodes : start_node and end_node, we also create two dictionaries , open_list_node with the nodes to explore and a visited_list with explored nodes.

With the library heapify we created a heap, open_list, which is a binary tree data structure, with for the first node the node with the lowest cost (f). Afterwards, we enter in a while structure and continue to execute the code as long as there are items in the open_list.

Then we use heapop to retrieve the first element of the list and to delete it, we call that element the current_node. We add this element to the visited_list and we check if the current_node is not the end_node.

If after some iterations of the loop while the current_node it's the end_node we return the path from the end_node. To return the pass we use the attribute parent of each node to pull-up the graph.

After we create the children's nodes from the current_node, we use the 8 possibilities (square+diag) around our current_node. Before to add each child to the list of children we check some conditions :

1. Check if the child is in range of the map (365,500).
2. Check if the terrain of the child is walkable (speed!=0).
3. Check if the child is in the dictionary visited_list.

If the child verified these conditions we add it to the list of children and we continue the algorithm.

Next, we have a list of children and for each child in the list we calculated the total cost function (f). The total cost function is equal to the cost function(g) + the heuristic function(h). For this problem we decided to work with the euclidean distance for the cost function and with the diagonal distance for the heuristic function :

$$f(n) = g(n) + h(n)$$

$$g(n) = g(current) + \sqrt{(|child.x - current.x| * length)^2 + (|child.y - current.y| * width)^2 + |child.z - current.z|} * \frac{1}{chidl.speed}$$

$$h(n) = (d_x + d_y + d_z - d_{min} - d_{max}) * Maxpseed \quad with\ d_x = |child.x - end.x| * length,$$
$$d_y = |child.y - end.y| * width,\ d_z = |child.z - end.z|,\ d_{min} = min(dx, dy)\ and\ d_{max} = max(d_x, d_y)$$

We also check if the child is already in the open_list_node and if he is and if he has a cost function lower than the previous one in the list. If it is the case, we retrieve the index of the previous and we update the parameters (h,g,f) of the previous.

If the child is not in the open_list_node we add the node to open_list and open_list_node with heap_push. And we continue to execute the while loop until find the end_node.

Each pixel of the image is represented with a unique key in order to quickly identify the node. We access the dictionary by using the key and we can check easily if the node is in the open_list_node or in the visited_list.

**Troubleshooting with the implementation (data structure, heuristic,cost function) :**

I started to use python list at the beginning with an heuristic function defined by the euclidean distance between the current_node and the end_node. I quickly understood that my data structure was bad. In fact, the complexity of the A* is exponential and we have to find the right data structure if we want to work on large scale examples such as the elevation one in the use case file.

So I decided to associate each node with a key in order to access the list stored in a dictionary. After that changement, I saved a lot of time when I was running my program.

All the test cases worked very well except the elevation one. In fact, the heuristic function of the program was too important in comparison to the cost function. The program overestimated the heuristic function and thus returned a wrong path that was much larger than the actual path that is supposed to follow the mountains.

Then I looked into it and saw that it was better to use diagonal distance as a heuristic function. I adapted this function to a 3-dimensional problem and I was able to run my program correctly on the proposed test boxes. The euclidean distance works well for small scale problems but here when we try to perform the elevation test case the cost function is totally underestimated compared to the heuristic.

**Test cases :**

All test cases have been done and worked after the last program changes. Normally, all distances match with the distances specified in the file. The times are respected except for the test case elevation which takes 140s and not <60s.

I think that my data structure is not good enough. I maybe should only use two dictionaries with a dictionary for the closed_node and an ordered dictionary for the open_node.

**The test case outputs are stored in each test case file with the name “xxxx_out.png”. The distance is also written on the output RGB image.**

**I used : OpenCV, Numpy, Heapq, Time and the Math functions.**