Sentral asprects of the implementation

The **search** class search contains general variable such as openlist, closedlist and states generated. Since the nodes of the algorithm are problem specific, we have chosen to have it as a seperate class and change its properties by calling functions such as **appendkid(), generateSuccessors()** and **compute heuristics**. Although the node class is abstract, implementation of it is easily achieveble. The algorithm runs with nodes that has all methods it uses within it such that the Search class which contains A* is independent of specific node types. The main flow is as followed in the psuedocode from the assignment, the agenda loop start by popping from the openlist, appended to the closedlist, and then proceeds with the check to find if the the node is the solution. Further it generates succesors and checking the conditions, newly uniquely generated nodes gets pushed into the openlist and already generated nodes gets updated. When solution is found the path is recursively constructed as a part of the algorithms condition test when it reach the goal.

-agenda loop
-how it can be easily reused

DEFINE **best-first-search**()

1. CLOSED ← ∅; OPEN ← ∅
2. Generate the initial node, n0, for the start state.
3. g(n0) ← 0; Calculate h(n0)
4. f(n0) ← g(n0) + h(n0);
5. Push n0 onto OPEN.
6. **Agenda Loop:** While no solution found do:
   - If OPEN = ∅ return FAIL
   - X ← pop(OPEN)
   - push(X,CLOSED)
   - If X is a solution, return (X, SUCCEED)
   - SUCC ← generate-all-successors(X)
   - For each S ∈ SUCC do:
     - If node S* has previously been created, and if state(S*) = state(S), then S ← S*.
     - push(S,kids(X))
     - If not(S ∈ OPEN) and not(S ∈ CLOSED) do:
       * **attach-and-eval**(S,X)
       * insert(S,OPEN) ;; OPEN is sorted by ascending f value.
     - else if g(X) + arc-cost(X,S) < g(S) then (found cheaper path to S):
       * **attach-and-eval**(S,X)
       * If S ∈ CLOSED then **propagate-path-improvements**(S)

Heuristic function

In this assignment the heuristic function used for the A* pathfinding implementation is the manhattan distance. The manhattan distance is computed by summing up the absolute value of both horizontal and vertical distances from a point P to the destination D. The downside of the manhattan distance is that it does not take obstacles into consideration.

Following is the implementation used, where the absolute value of the horizontal distance and vertical distance from the chosen point to the goal are summed up.

```
# HEURISTIC
def calculate_heuristic(self , position, goal):
    return abs(position[0]- goal[0]) + abs(position[1]-goal[1])
```

How successors are generated

The successors are generated by a function called **generate_successors**, and they are all of the Node class. The figure below shows how the north successor is generated. Other successors are generated in the same way. They key point is that successors are generated if they don't violate the constraint of either being a position out of the map or a wall. All successors are generated, but only the ones being new (not in open or closed- list) will be added to the openlist via the **attach_eval** function. Since the problem is simple the nodes in the algorithm do not get the state variable updated, but it is available other problems.

```
#north
if node.position[1]+1 <= self.map.mapsize[1]-1:
    if (node.position[0], node.position[1]+1) not in self.map.walls:
        new_node3 = Node((node.position[0], node.position[1]+1), node)
        successors.append(new_node3)
```