Machine Exercise #1

Solution by Searching

```
5 5
1 1 1 1 1
0 1 1 1 1
1 1 0 0 1
1 1 0 0 0
1 1 1 1 0
Source: 2 2
Destination: 4 4
Up: 4
Down 4
Left: 5
Right: 5
Diagonal: 6
```

Sample Input

1. On the right is a sample input. The program will parse the input first. The map will be stored a 2D list like this :

   [[1,1,1,1,1], [0,1,1,1,1],[1,1,0,1,1],[1,1,0,0,0],[1,1,1,1,0]]

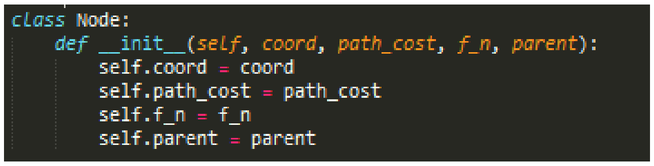
2. The program will convert the map into an adjacency dictionary. The Key for the dictionary is the coordinate. Its value is a list of coordinates that is adjacent to the key.

   { 1 0 : [],
   2 2 : [2 3, 3 2, 3 3],
   2 3 : [2 2, 3 2, 3 3, 3 4],
   3 2 : [2 2, 2 3, 3 3],
   3 3 : [2 2, 2 3, 3 2, 3 4, 4 4],
   3 4 : [2 4, 3 3, 4 4],
   4 4 : [3 3, 3 4]}

3. A dictionary is also created to store heuristic values.

   { 1 0 : 5
   2 2 : 2.828
   2 3 : 2.236
   3 2 : 2.236
   3 3 : 1.414
   3 4 : 1
   4 4 : 0 }

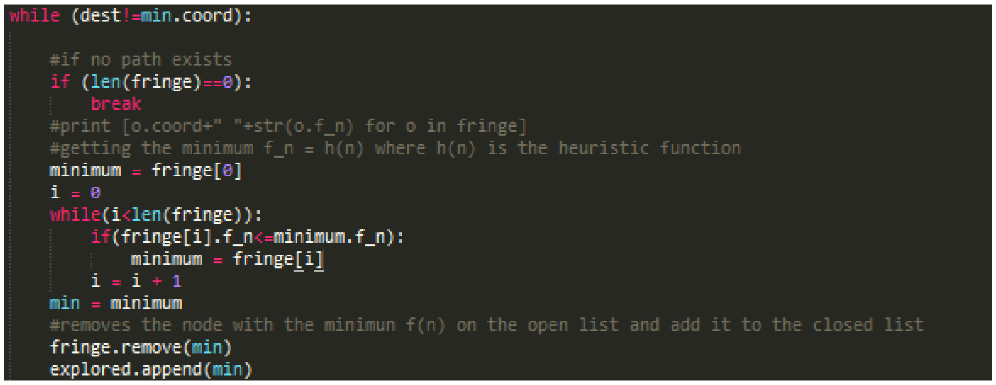
4. A class named "Node" is created. Each node has a **coordinate, path_cost, f(n), and parent node.** f(n) is the evaluation function. It is equal to h(n) in greedy best-first search where h(n) is the heuristic function. It is equal to path cost + h(n) in A* search.



```python
class Node:
    def __init__(self, coord, path_cost, f_n, parent):
        self.coord = coord
        self.path_cost = path_cost
        self.f_n = f_n
        self.parent = parent
```

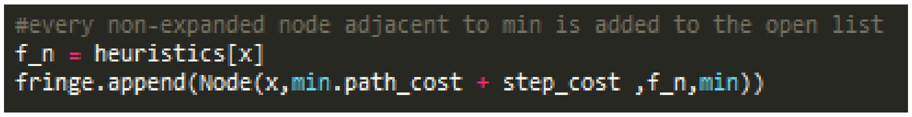
5. Two lists are initialized to be empty. The **fringe** (open list / nodes that are not yet expanded) and the **expanded** (closed list).

6. The start node is first inserted into the open list. Then, the f(n) of the nodes in the open list will be compared. The node with the lowest f(n) will be removed from the open list and transferred to the closed list.



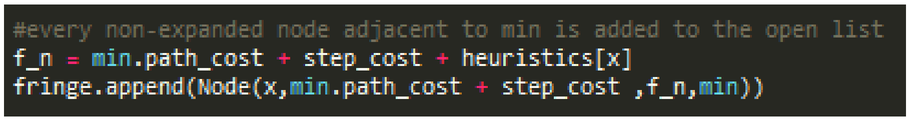
```python
while (dest!=min.coord):

    #if no path exists
    if (len(fringe)==0):
        break
    #print [o.coord+" "+str(o.f_n) for o in fringe]
    #getting the minimum f_n = h(n) where h(n) is the heuristic function
    minimum = fringe[0]
    i = 0
    while(i<len(fringe)):
        if(fringe[i].f_n<=minimum.f_n):
            minimum = fringe[i]
        i = i + 1
    min = minimum
    #removes the node with the minimun f(n) on the open list and add it to the closed list
    fringe.remove(min)
    explored.append(min)
```

7. Then all unexpanded nodes adjacent to the node with the lowest f(n) will be inserted into the open list.
   For **greedy best-first search**, the third argument, f(n), is equal to heuristics[x].



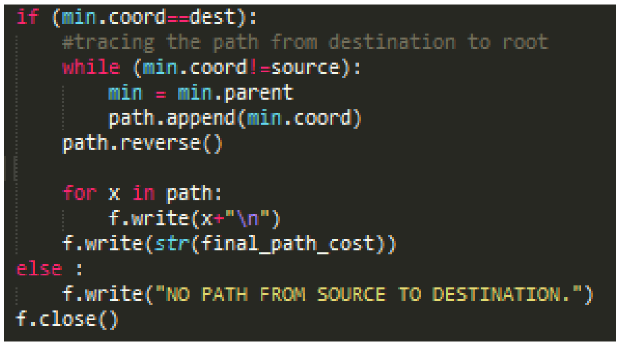
```python
#every non-expanded node adjacent to min is added to the open list
f_n = heuristics[x]
fringe.append(Node(x,min.path_cost + step_cost ,f_n,min))
```

   In **A* search**, the third argument, f(n), is equal to heuristics[x] + new path cost (where new path cost = old path cost + step cost)



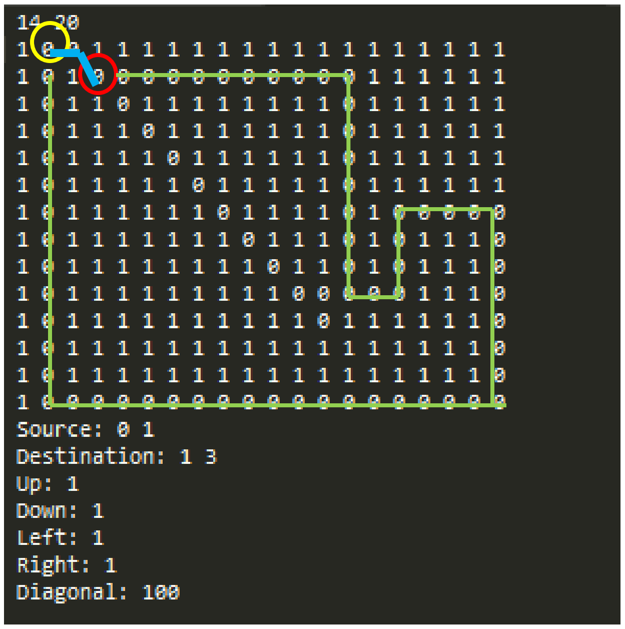
```python
#every non-expanded node adjacent to min is added to the open list
f_n = min.path_cost + step_cost + heuristics[x]
fringe.append(Node(x,min.path_cost + step_cost ,f_n,min))
```

8. The step cost will be computed. The path cost per node in the open list will be updated.
9. The program will compare again the f(n) values of the nodes in the open list. The node with the minimum f(n) value will be removed from the open list and will be transferred to the closed list. All unexpanded nodes adjacent to the transferred node will be added to the open list. Their path cost and f(n) values will be updated.
10. The algorithm will terminate if the destination coordinate is selected as the minimum on the open list. If the open list becomes empty, yet the destination coordinate is not yet found, the algorithm terminates and will output “no path found”
11. The program will “backtrack” from destination to source to determine the path. Since the node’s parent is stored in every node, the source can easily be reached using a loop.



```python
if (min.coord==dest):
    #tracing the path from destination to root
    while (min.coord!=source):
        min = min.parent
        path.append(min.coord)
    path.reverse()

    for x in path:
        f.write(x+"\n")
    f.write(str(final_path_cost))
else :
    f.write("NO PATH FROM SOURCE TO DESTINATION.")
f.close()
```

12. Sample input and output



```
14 20
1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
1 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1
1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1
1 0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 0 0
1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0
1 0 1 1 1 1 1 1 1 1 0 1 1 0 1 0 1 1 1 0
1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0
1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0
1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Source: 0 1
Destination: 1 3
Up: 1
Down: 1
Left: 1
Right: 1
Diagonal: 100
```

In this example, the source is encircled yellow destination is encircled red. The destination can be easily reached through the blue line but the cost is high because a diagonal movement cannot be avoided.

The greedy search ouputs the blue line path with a cost of 101.0. While the A* search ouputs the green line path with a cost of 65.0.