Pattarapon Buathong 62070504012 Automation Engineering

INC491 HOMEWORK 1 Report

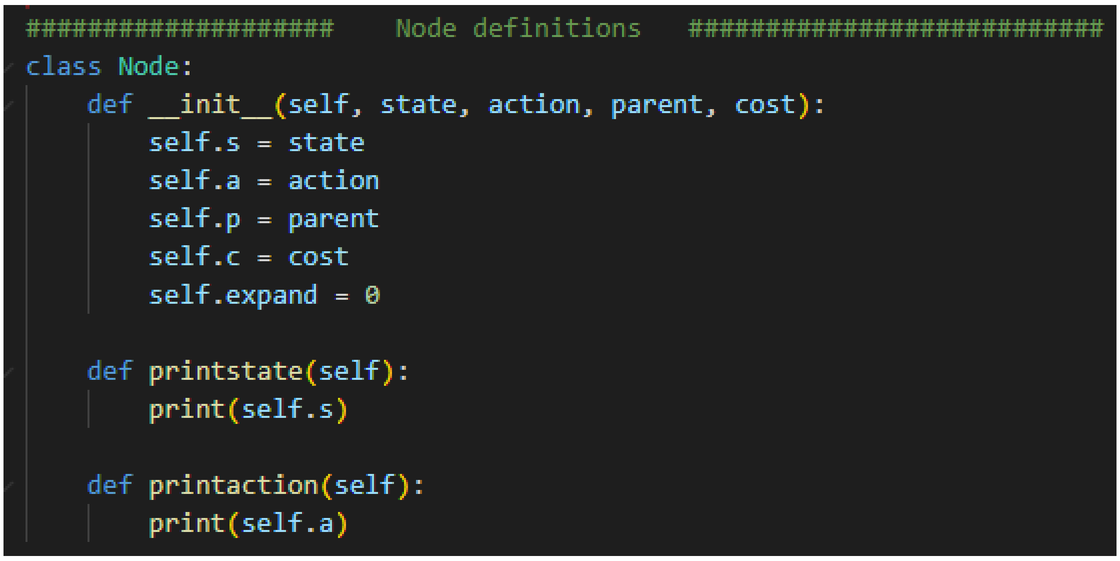
1. Explanation of the BFS source code in each part and answer 4 questions below.

Explanation of BFS source code

```python
import numpy as np
```

The Python library NumPy is used to manipulate arrays. Additionally, it has matrices, Fourier transform, and functions for working in the area of linear algebra. We import numpy library as np so that we don't have to write numpy every time we want to use its functions.

<u>Node Definitions</u>



```python
####################    Node definitions    ###########################
class Node:
    def __init__(self, state, action, parent, cost):
        self.s = state
        self.a = action
        self.p = parent
        self.c = cost
        self.expand = 0

    def printstate(self):
        print(self.s)

    def printaction(self):
        print(self.a)
```

The object-oriented class named Node is used to define each node in a search tree.

The __init__ method lets the class initialize the object's attributes.

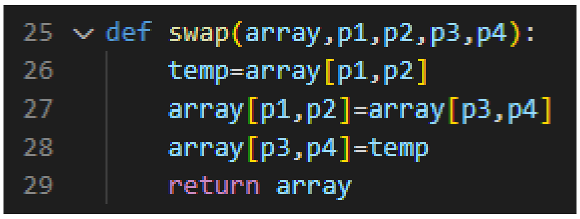
- State: used to define the states of the node according to the action functions, i.e., the whole 3x3 sliding puzzle at the moment
- Action: the action of the node (up, down, left, right)
- Parent: the parent of the node
- Cost: the depth of search tree used in the operation
- Expand: Initialization of expand variable is equal to 0

The printstate function prints the state of the Node.

The printaction function prints the action of the Node.

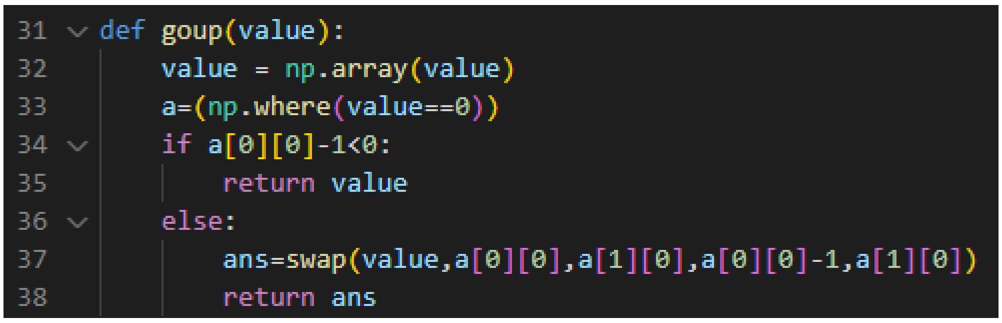
<u>Action Functions</u>

The action functions consist of 5 different Python function, they are swap, go up, go down, go left, and go right. The swap function is used in other functions.



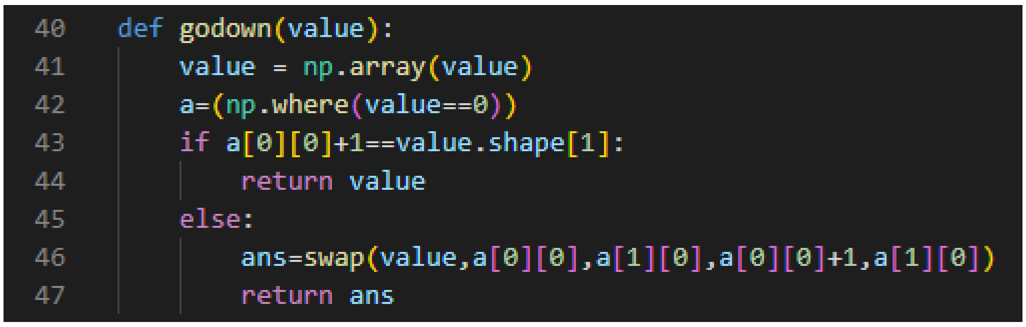
```python
def swap(array,p1,p2,p3,p4):
    temp=array[p1,p2]
    array[p1,p2]=array[p3,p4]
    array[p3,p4]=temp
    return array
```

The swap function is used to swap the position between two elements in the array. First, it stores the current position value in the temp (temporary) variable. Next, it replaces the current position value with the position that we wanted to swap with. After that, the position that we wanted to swap with is replaced by the initial position value we stored in the temp variable.



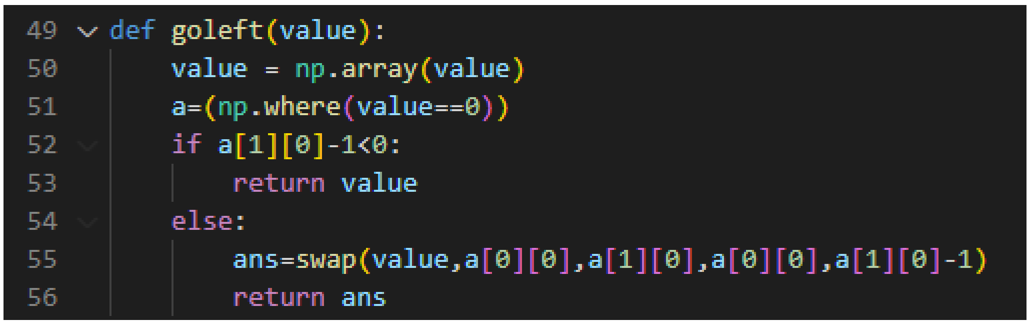
```python
def goup(value):
    value = np.array(value)
    a=(np.where(value==0))
    if a[0][0]-1<0:
        return value
    else:
        ans=swap(value,a[0][0],a[1][0],a[0][0]-1,a[1][0])
        return ans
```

The goup function (go up) swaps the “0” value with the value that is on the top of it within the array. First, it receives the value in the function (which is the state of the parent node). Variable “a” will store the row and column position of wherever the “0” is. Then, it checks whether a[0][0]-1 is less than 0 (a[0][0] in this case is the row of the position, so if the row is 0 (top row), it cannot go up anymore). If not, it will swap the 0 to the top, and other to the initial position of 0.



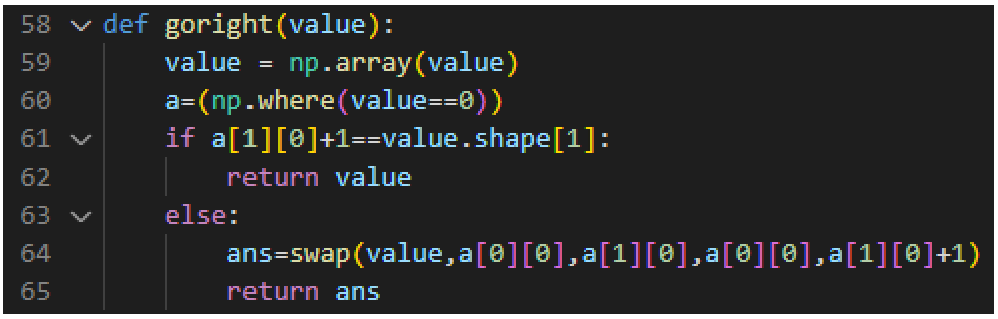
```python
def godown(value):
    value = np.array(value)
    a=(np.where(value==0))
    if a[0][0]+1==value.shape[1]:
        return value
    else:
        ans=swap(value,a[0][0],a[1][0],a[0][0]+1,a[1][0])
        return ans
```

The godown function (go down) swaps the “0” value with the value that is on the bottom of it within the array. First, it receives the value in the function (which is the state of the parent node). Variable “a” will store the row and column position of wherever the “0” is. Then, it checks whether a[0][0]+1 is equal to value.shape[1] (which is 3), meaning that 0 is at the last row, it cannot proceed to go down further. If not, it will swap the 0 to the bottom, and other to the initial position of 0.



```python
def goleft(value):
    value = np.array(value)
    a=(np.where(value==0))
    if a[1][0]-1<0:
        return value
    else:
        ans=swap(value,a[0][0],a[1][0],a[0][0],a[1][0]-1)
        return ans
```

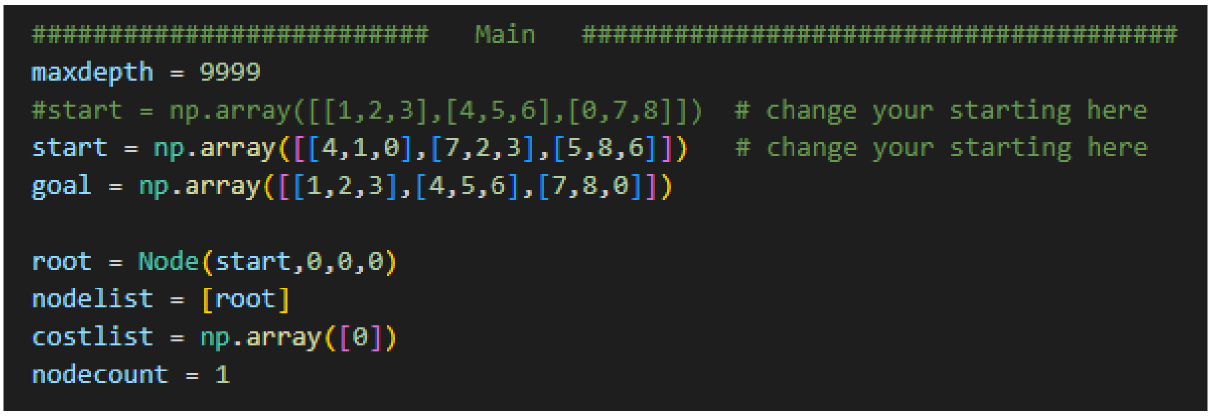
The goleft function (go left) swaps the 0 value with the value that is on the left of it within the array. First, it receives the value in the function (which is the state of the parent node). Variable "a" will store the row and column position of wherever the "0" is. Then, it checks whether a[1][0]-1 is less than 0 (a[1][0] is the column position of the "0", if a[1][0]-1 < 0 it mean that the "0" is on the leftmost column of the array, meaning that it cannot go left further). If not, it will swap the 0 to the left, and other to the initial position of 0.



```python
def goright(value):
    value = np.array(value)
    a=(np.where(value==0))
    if a[1][0]+1==value.shape[1]:
        return value
    else:
        ans=swap(value,a[0][0],a[1][0],a[0][0],a[1][0]+1)
        return ans
```

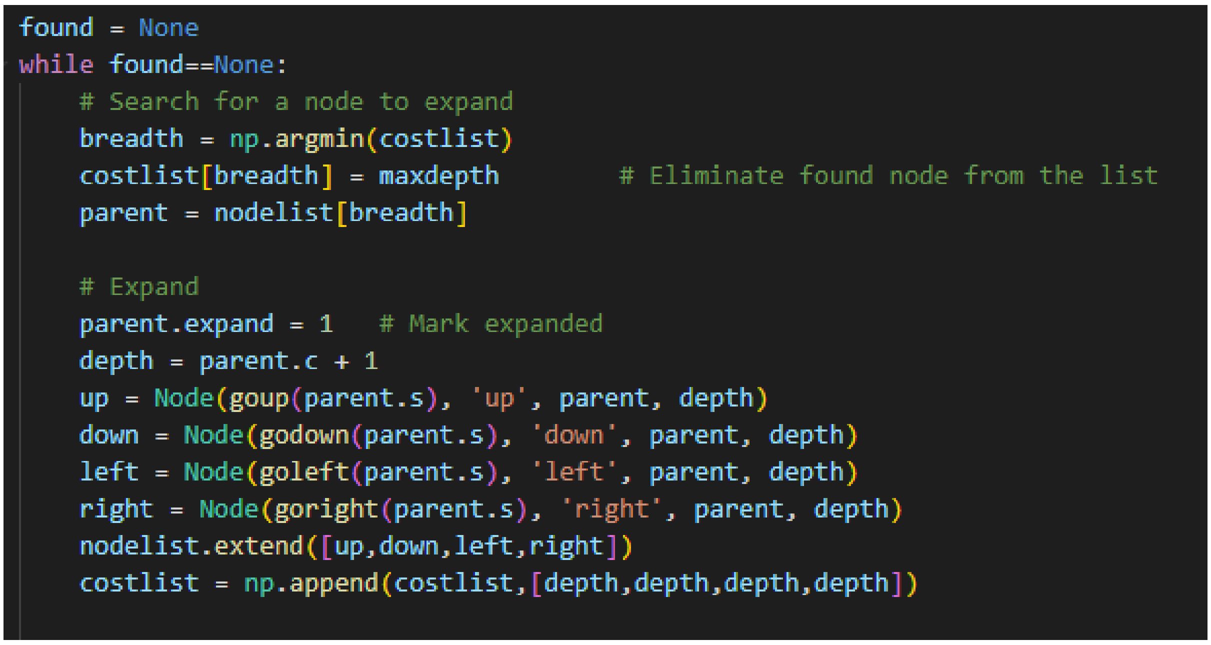
The goright function (go right) swaps the 0 value with the value that is on the right of it within the array. First, it receives the value in the function (which is the state of the parent node). Variable "a" will store the row and column position of wherever the "0" is. Then, it checks whether a[1][0]+1 is equal to value.shape[1] (or 3) (a[1][0] is the column position of the "0" in the array, if a[1][0]+1 is equal to 3, it mean that the "0" is on the rightmost column of the array, meaning that it cannot go right further). If not, it will swap the 0 to the right, and other to the initial position of 0.

Main Code



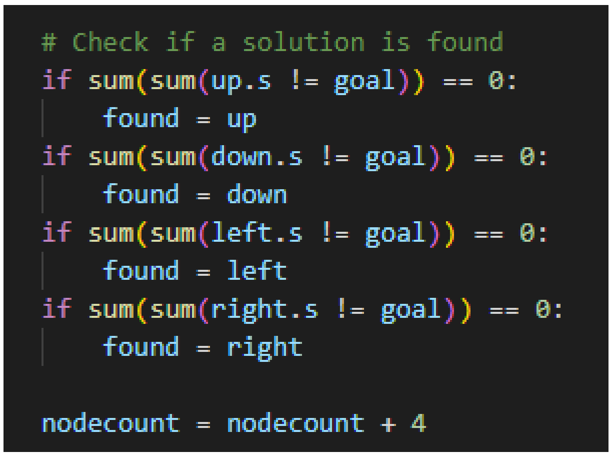
```python
##########################   Main   #######################################
maxdepth = 9999
#start = np.array([[1,2,3],[4,5,6],[0,7,8]])  # change your starting here
start = np.array([[4,1,0],[7,2,3],[5,8,6]])   # change your starting here
goal = np.array([[1,2,3],[4,5,6],[7,8,0]])

root = Node(start,0,0,0)
nodelist = [root]
costlist = np.array([0])
nodecount = 1
```

Maxdepth variable declares the maximum level that the search tree will declines down. Start variable is the initial array. The goal variable is the goal array of the sliding puzzle that we want. Root variable declares the topmost initial node. Nodelist variable stores the node in a list, so we can keep the state, action, parent, and cost of each node. Costlist variable stores the depth of the search as an array, it will be appended with depth variable later. Nodecount is used to store the number of nodes.



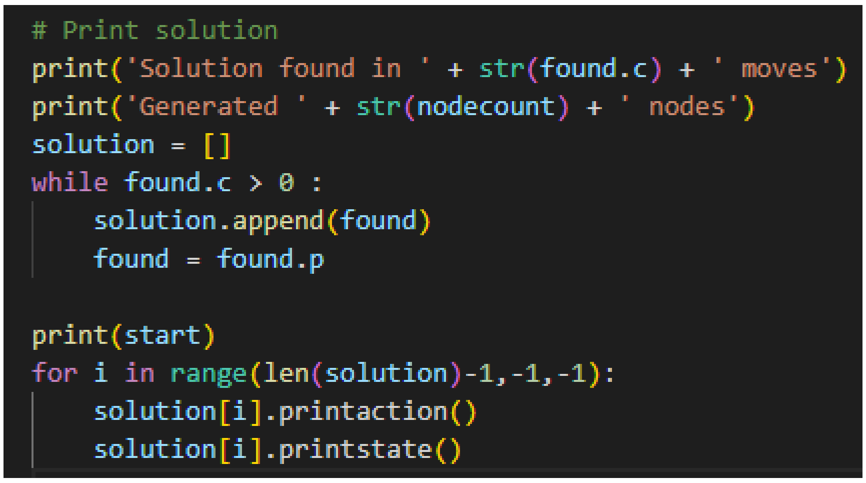
```python
found = None
while found==None:
    # Search for a node to expand
    breadth = np.argmin(costlist)
    costlist[breadth] = maxdepth        # Eliminate found node from the list
    parent = nodelist[breadth]

    # Expand
    parent.expand = 1   # Mark expanded
    depth = parent.c + 1
    up = Node(goup(parent.s), 'up', parent, depth)
    down = Node(godown(parent.s), 'down', parent, depth)
    left = Node(goleft(parent.s), 'left', parent, depth)
    right = Node(goright(parent.s), 'right', parent, depth)
    nodelist.extend([up,down,left,right])
    costlist = np.append(costlist,[depth,depth,depth,depth])
```

The found initial value is None, it will be changed when the solution is found. The while loop condition is do if found variable is still None. Breadth is the value of np.argmin that returns the indices of the minimum values along an axis. Costlist stores maxdepth in the position of “breadth” of the array. Parent variable is the value of nodelist at the position of breadth. The initial will be nodelist[0]. The expand starts by changing the expand variable of the parent from 0 to 1. Then, depth is equal to the cost of the parent node + 1. The up variable is a node which have the state of go up of its parent’ state. The down variable is a node which have the state of go down of its parent’ state. The left variable is a node which have the state of go left of its parent’ state. The right variable is a node which have the state of go right of its parent’ state. Then, the nodelist is extended (Python command). The extend() method adds all the elements of an iterable (list, tuple, string etc.) to the end of the list. Then, costlist is appended with [depth, depth, depth, depth] (up, down, left, right depths).



```python
# Check if a solution is found
if sum(sum(up.s != goal)) == 0:
    found = up
if sum(sum(down.s != goal)) == 0:
    found = down
if sum(sum(left.s != goal)) == 0:
    found = left
if sum(sum(right.s != goal)) == 0:
    found = right

nodecount = nodecount + 4
```

After that, we check if a solution is found. The up.s != goal match the state in the “up”, if every elements is exactly the same as the goal, all will be replaced with zero, if not, 1. So, if all is 0, the sum will be 0, and the if case will be true, meaning that we found the solution in the up state. This is the same in every variable (up, down, left, right). If nothing is equal to 0, it means that we haven't found the solution yet. Then, the nodecount will be added up by 4.



```
# Print solution
print('Solution found in ' + str(found.c) + ' moves')
print('Generated ' + str(nodecount) + ' nodes')
solution = []
while found.c > 0 :
    solution.append(found)
    found = found.p

print(start)
for i in range(len(solution)-1,-1,-1):
    solution[i].printaction()
    solution[i].printstate()
```

If the solution is found, we will print the solution. We print how many moves it takes to find the solution (number of cost). How many nodes our code generated (nodecount). The empty list is declared to store the solution. While found's cost > 0 (the code actually did some searches, not start = goal as initial), it will append the found to the solution variable. Lastly, the start array is printed, as well as the actions and states to get the solution.

Questions

1.1 What datatype is the variable 'Node'?

→ It is an object. As it is written using Python class using object-oriented programming logic.

1.2 What does function 'swap' do?

→ The swap function is used to swap the position between two elements in the array. First, it stores the current position value in the temp (temporary) variable. Next, it replaces the current position value with the position that we wanted to swap with. After that, the position that we wanted to swap with is replaced by the initial position value we stored in the temp variable.

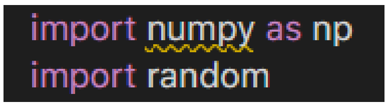
1.3 What does variable 'nodelist' do?

→ Nodelist variable stores the node in a list, so we can keep the state, action, parent, and cost of each node.

1.4 What does variable 'costlist' do?

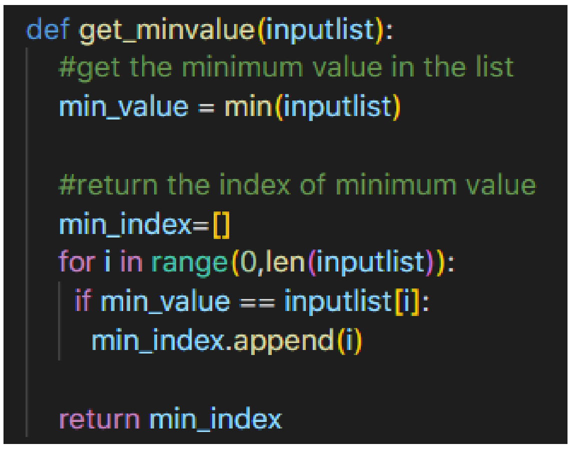
→ Costlist variable stores the depth of the search as an array, it will be appended with depth variable later.

2. Modify the code to use A* search. This should reduce the number of nodes a lot. Use heuristic h2 (Manhattan distance) that you learned in the class. Print out your source code and explain what has been modified by you.



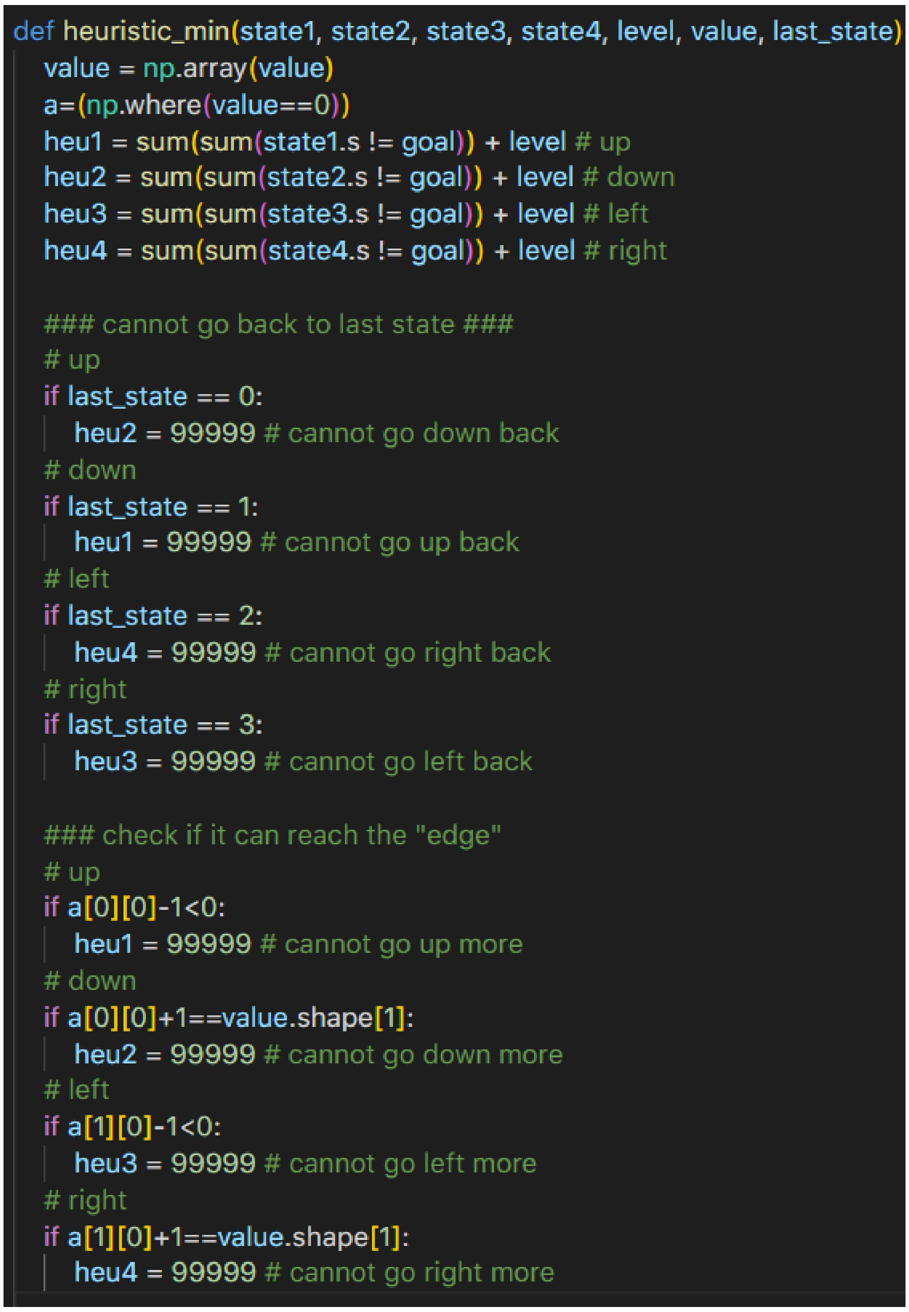
```python
import numpy as np
import random
```

I imported one more library, which is random, I will explain the reason later, in this section.

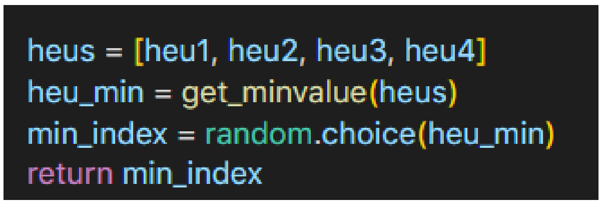


```python
def get_minvalue(inputlist):
    #get the minimum value in the list
    min_value = min(inputlist)

    #return the index of minimum value
    min_index=[]
    for i in range(0,len(inputlist)):
        if min_value == inputlist[i]:
            min_index.append(i)

    return min_index
```

This function is used to get a list of minimum value, in case of there are multiple minimum value, it will return all of the index, so we can use it to perform random later.

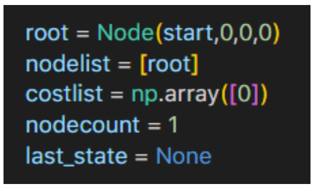


```python
def heuristic_min(state1, state2, state3, state4, level, value, last_state):
    value = np.array(value)
    a=(np.where(value==0))
    heu1 = sum(sum(state1.s != goal)) + level # up
    heu2 = sum(sum(state2.s != goal)) + level # down
    heu3 = sum(sum(state3.s != goal)) + level # left
    heu4 = sum(sum(state4.s != goal)) + level # right

    ### cannot go back to last state ###
    # up
    if last_state == 0:
        heu2 = 99999 # cannot go down back
    # down
    if last_state == 1:
        heu1 = 99999 # cannot go up back
    # left
    if last_state == 2:
        heu4 = 99999 # cannot go right back
    # right
    if last_state == 3:
        heu3 = 99999 # cannot go left back

    ### check if it can reach the "edge"
    # up
    if a[0][0]-1<0:
        heu1 = 99999 # cannot go up more
    # down
    if a[0][0]+1==value.shape[1]:
        heu2 = 99999 # cannot go down more
    # left
    if a[1][0]-1<0:
        heu3 = 99999 # cannot go left more
    # right
    if a[1][0]+1==value.shape[1]:
        heu4 = 99999 # cannot go right more
```



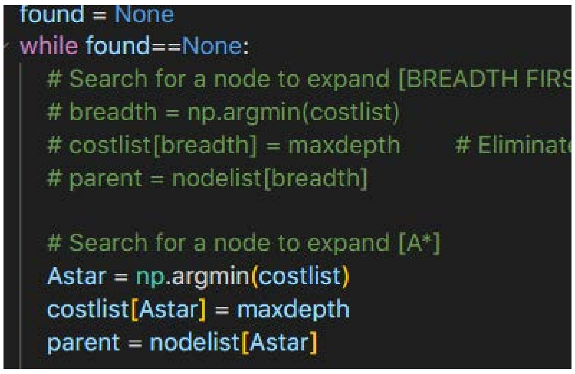
```python
heus = [heu1, heu2, heu3, heu4]
heu_min = get_minvalue(heus)
min_index = random.choice(heu_min)
return min_index
```

This function is used to calculate the minimum evaluation function to decide which node to expand. It sums up the heuristic and the cumulative cost of each action possible. It also checks two types of cases. First, to make sure it doesn't go back to its last state, I store the last state in last_state variable, for instance, if last state is "up". Now, it cannot go "down" to get back to its last state (make heu2 (for down) very high). Second, to make sure it doesn't go exceed the sliding puzzle array, this uses the same logic as in the action functions. We store every heuristic in the list and find the minimum value. If there are multiple minimum value, Python random will choose the choice from the list.



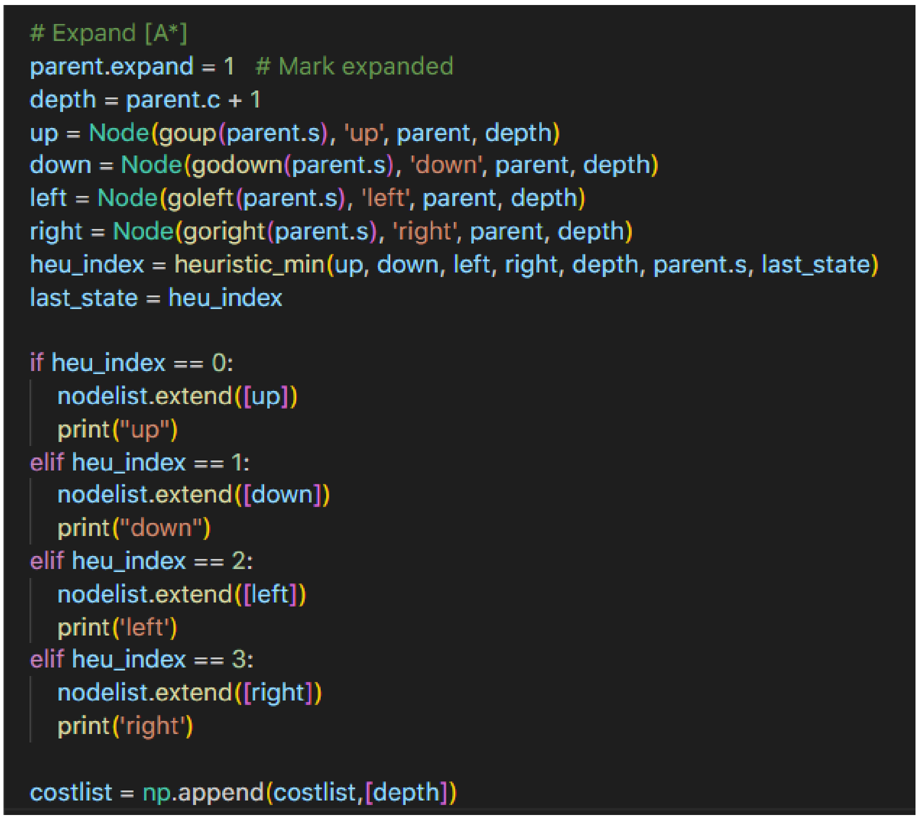
```
root = Node(start,0,0,0)
nodelist = [root]
costlist = np.array([0])
nodecount = 1
last_state = None
```

Here, I declared the last_state variable's initial value as None.



```
found = None
while found==None:
    # Search for a node to expand [BREADTH FIRS
    # breadth = np.argmin(costlist)
    # costlist[breadth] = maxdepth       # Eliminat
    # parent = nodelist[breadth]

    # Search for a node to expand [A*]
    Astar = np.argmin(costlist)
    costlist[Astar] = maxdepth
    parent = nodelist[Astar]
```

To avoid confusion, I create Astar variable to store argmins.



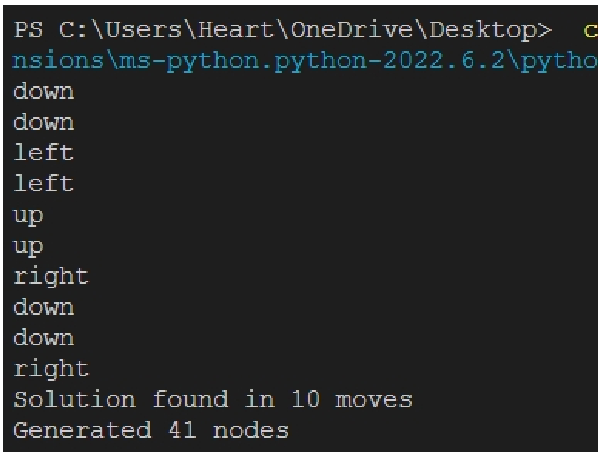
```
# Expand [A*]
parent.expand = 1   # Mark expanded
depth = parent.c + 1
up = Node(goup(parent.s), 'up', parent, depth)
down = Node(godown(parent.s), 'down', parent, depth)
left = Node(goleft(parent.s), 'left', parent, depth)
right = Node(goright(parent.s), 'right', parent, depth)
heu_index = heuristic_min(up, down, left, right, depth, parent.s, last_state)
last_state = heu_index

if heu_index == 0:
    nodelist.extend([up])
    print("up")
elif heu_index == 1:
    nodelist.extend([down])
    print("down")
elif heu_index == 2:
    nodelist.extend([left])
    print('left')
elif heu_index == 3:
    nodelist.extend([right])
    print('right')

costlist = np.append(costlist,[depth])
```

The processes are nearly the same as Breadth-first search, but instead of extending to every actions. Now we only extend to the path with minimum evaluation function.

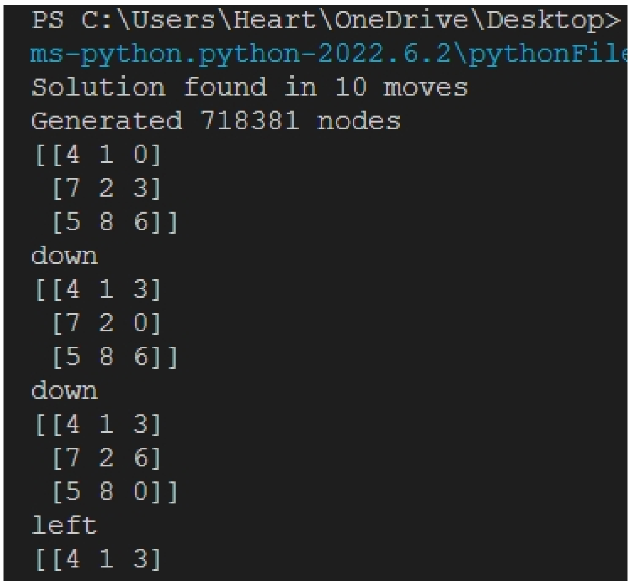
3. Test the performance of A* by comparing it with the BFS. Make initial positions that require solutions with different number of moves. Record the time used and the number of nodes generated for both BFS and A* like in this table. Do not wait for more than 15 minutes. Make the table as long as you see appropriate.

Depth	BFS	A*
8	8 sec (63021 nodes)	<1 sec (33 nodes)
9	17 sec (194093 nodes)	1 sec (37 nodes)
10	2 min 21 sec (718381 nodes)	3 sec (41 nodes)

I had waited for more than 20 minutes, and BFS still not finished searching: start = np.array([[4,0,1],[7,2,3],[5,8,6]]) with depth = 11 which is only one more simple move from the original start = np.array([[4,1,0],[7,2,3],[5,8,6]]) that has depth = 10.


```
PS C:\Users\Heart\OneDrive\Desktop>  c
nsions\ms-python.python-2022.6.2\pytho
down
down
left
left
up
up
right
down
down
right
Solution found in 10 moves
Generated 41 nodes
```

A* Search


```
PS C:\Users\Heart\OneDrive\Desktop>
ms-python.python-2022.6.2\pythonFile
Solution found in 10 moves
Generated 718381 nodes
[[4 1 0]
 [7 2 3]
 [5 8 6]]
down
[[4 1 3]
 [7 2 0]
 [5 8 6]]
down
[[4 1 3]
 [7 2 6]
 [5 8 0]]
left
[[4 1 3]
```

Breadth-first search

In conclusion, BFS time used to search will drastically increase if the depth increased. Comparing BFS with A* search, we can see that A* is a lot faster than BFS, since it did not have to search every node in the depth like BFS.