# Artificial Intelligence

| Name | ID |
|---|---|
| Malak Mahmoud Medhat Aref | 20221445867 |
| Nureen Ehab Mahmoud Barakat | 20221465124 |
| Zainab Mohamed Abdallah | 20220131251 |

# Using Informed and Uninformed Search Algorithms to Solve 8-Puzzle

- We will implement the 8-puzzle search problem using the three search algorithms: BFS, DFS and A*.

- For implementation we will use python.

- Our program contains the following:

    1. Three classes for each search algorithm.

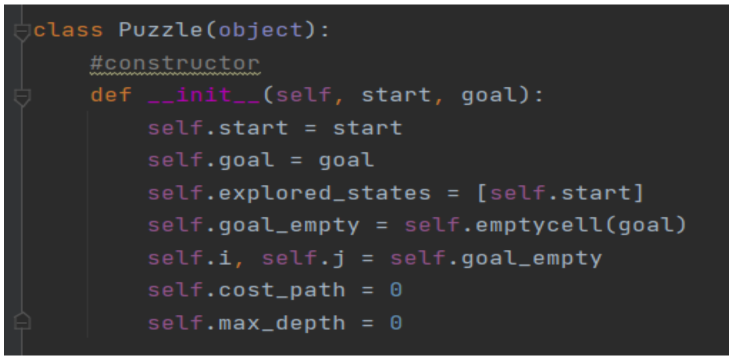
    2. A class named Board contains some variables (the state, previous board, direction, cost, heuristic).

    3. A class named Puzzle contains all the common functions and variables between the 3 search classes except for the functions (heuristic, get_next) belongs to A*.

- First, we will discuss the Puzzle class and then we will talk about each class separately.

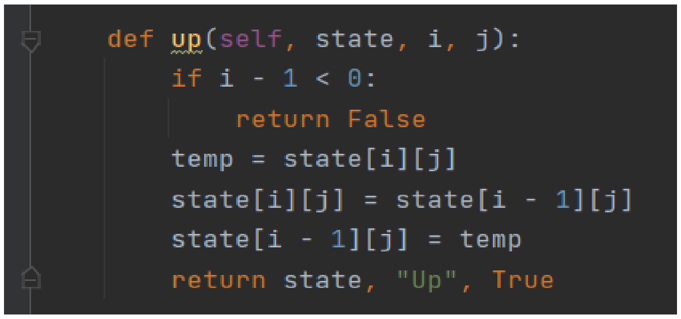
## The Puzzle Class contains the following:

1. We have a constructor that contains start and goal as parameters and then we initialize them.



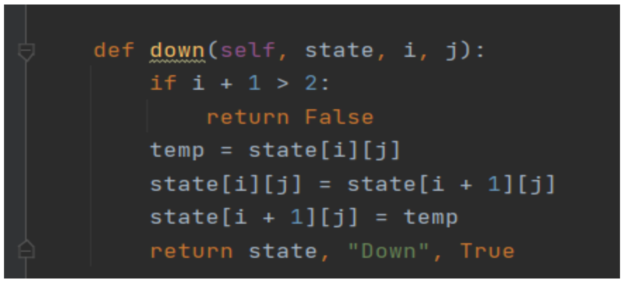
```
class Puzzle(object):
    #constructor
    def __init__(self, start, goal):
        self.start = start
        self.goal = goal
        self.explored_states = [self.start]
        self.goal_empty = self.emptycell(goal)
        self.i, self.j = self.goal_empty
        self.cost_path = 0
        self.max_depth = 0
```

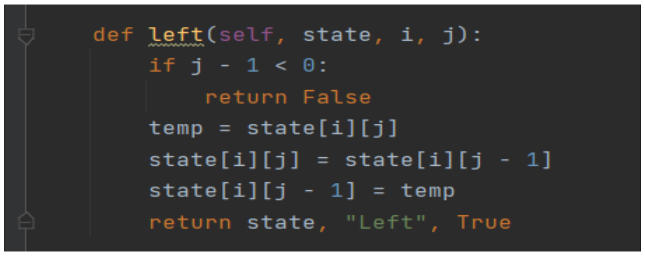
2. Define a function called "**Up**" takes (state, i, j) as parameters. We pretend that i is equivalent to x and j is equivalent to y. So, we made an if condition to check whether the move is inside the border or not. If (i-1<0) return false.
   because there will be error in the border because the border length is 3*3 and the indexes are (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)
   else we created a swap

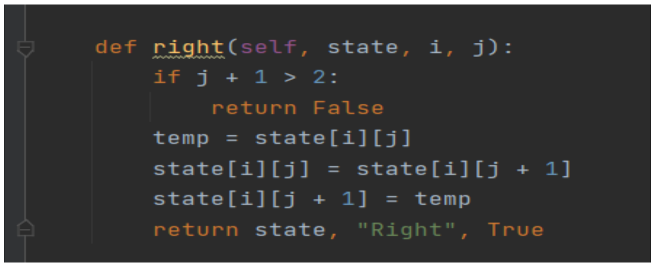


```
    def up(self, state, i, j):
        if i - 1 < 0:
            return False
        temp = state[i][j]
        state[i][j] = state[i - 1][j]
        state[i - 1][j] = temp
        return state, "Up", True
```

and return the state, the move which is Up and true.

3. Define another function called “**Down**” equivalent to the function up but with a different if condition
   If (i+1>2) return false.
   else we created a swap
   
   ```
   def down(self, state, i, j):
       if i + 1 > 2:
           return False
       temp = state[i][j]
       state[i][j] = state[i + 1][j]
       state[i + 1][j] = temp
       return state, "Down", True
   ```
   and return the state, the move which is Down and true.

4. Define another function called “**Left**” equivalent to the function up but with a different if condition
   If (j-1<0) return false.
   else we created a swap
   
   ```
   def left(self, state, i, j):
       if j - 1 < 0:
           return False
       temp = state[i][j]
       state[i][j] = state[i][j - 1]
       state[i][j - 1] = temp
       return state, "Left", True
   ```
   and return the state, the move which is Left and true.

5. Define another function called “**Right**” equivalent to the function up but with a different if condition
   If (j+1>2) return false.
   else we created a swap
   
   ```
   def right(self, state, i, j):
       if j + 1 > 2:
           return False
       temp = state[i][j]
       state[i][j] = state[i][j + 1]
       state[i][j + 1] = temp
       return state, "Right", True
   ```
   and return the state, the move which is Right and true.

6. Define a function called **empty_cell** to return the index (i, j) of the empty tile.

7. Define a function called **solved** to check if we reached the goal by comparing the state with goal.

8. Define a function called **explored** to check whether visited this state before or no to prevent repetition**.**

9. Define a function called **heuristic** for A* search containing **Manhattan heuristic** and **Euclidean heuristic** and trying both we will compare between number of nodes expanded and output paths, and to report which heuristic is more admissible.
   - Manhattan heuristic
     It is the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively,
     h = abs (current_cell:x - goal:x) + abs (current_cell:y - goal:y)
   - Euclidean Distance
     It is the distance between the current cell and the goal cell using the distance formula=
     sqrt ((current_cell:x - goal:x)$^2$+(current_cell:y - goal:y)$^2$)

10. Define a function called **get_next** to get the least cost of the boards by making loop on all boards and finding the minimum cost
    (cost= depth + heuristic) then the least cost board will generate from it.

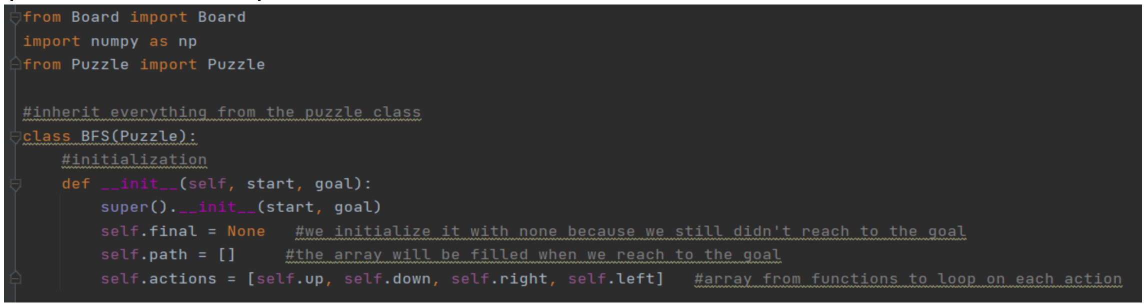
11. Define a function called **generate_path** will take the final puzzle that we reached and go backward till the start puzzle then will put all the boards in array.

12. Define a function called **print_path** takes array from generate path and print it.

# <u>The BFS Class contains the following:</u>

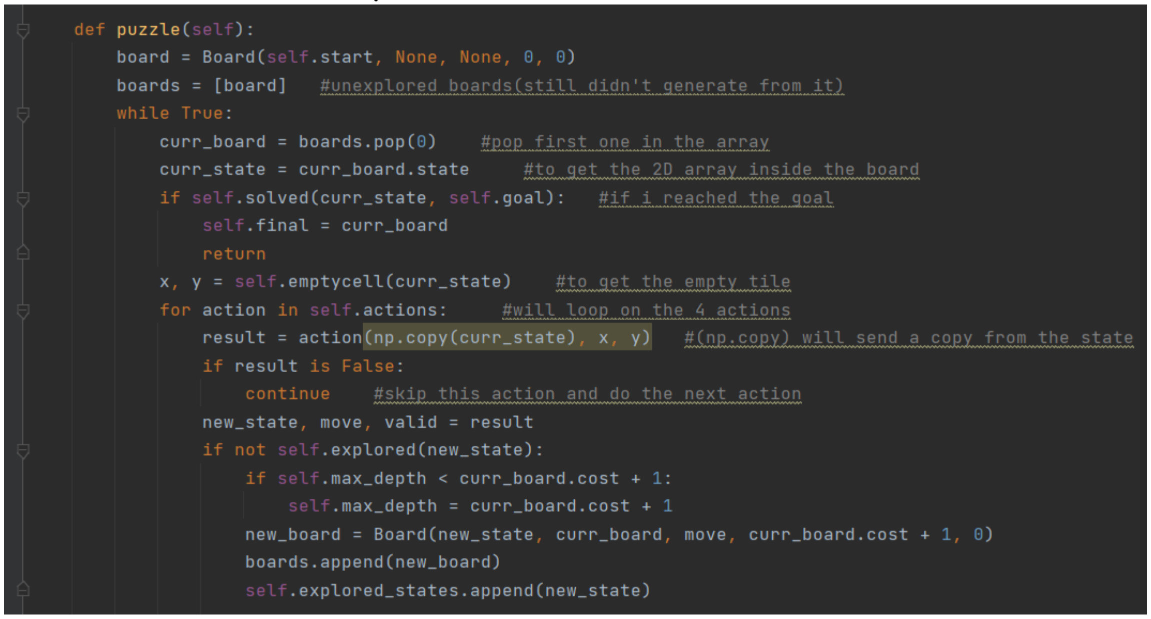
For implementation will use queue. The BFS expand shallowest node.

1. The BFS class will inherit all the functions from the Super class which is (The Puzzle class).



```python
from Board import Board
import numpy as np
from Puzzle import Puzzle

#inherit everything from the puzzle class
class BFS(Puzzle):
    #initialization
    def __init__(self, start, goal):
        super().__init__(start, goal)
        self.final = None   #we initialize it with none because we still didn't reach to the goal
        self.path = []     #the array will be filled when we reach to the goal
        self.actions = [self.up, self.down, self.right, self.left]   #array from functions to loop on each action
```

2. Define a function called puzzle



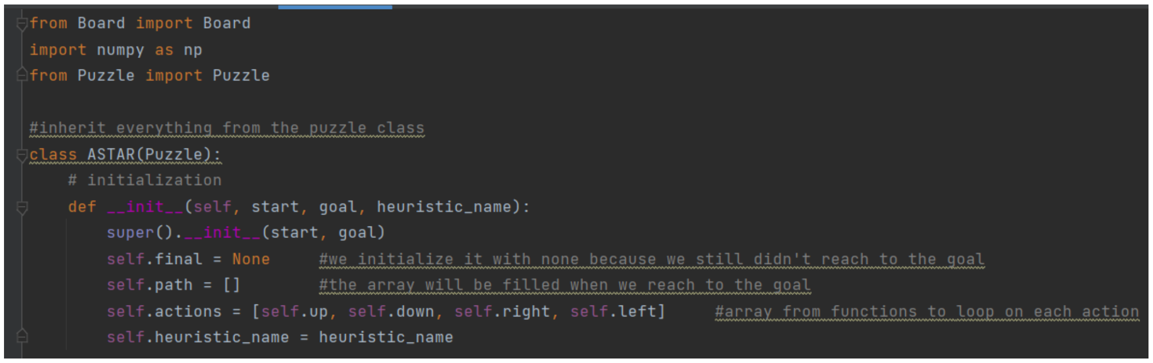
```python
    def puzzle(self):
        board = Board(self.start, None, None, 0, 0)
        boards = [board]   #unexplored boards(still didn't generate from it)
        while True:
            curr_board = boards.pop(0)    #pop first one in the array
            curr_state = curr_board.state      #to get the 2D array inside the board
            if self.solved(curr_state, self.goal):   #if i reached the goal
                self.final = curr_board
                return
            x, y = self.emptycell(curr_state)    #to get the empty tile
            for action in self.actions:      #will loop on the 4 actions
                result = action(np.copy(curr_state), x, y)   #(np.copy) will send a copy from the state
                if result is False:
                    continue    #skip this action and do the next action
                new_state, move, valid = result
                if not self.explored(new_state):
                    if self.max_depth < curr_board.cost + 1:
                        self.max_depth = curr_board.cost + 1
                    new_board = Board(new_state, curr_board, move, curr_board.cost + 1, 0)
                    boards.append(new_board)
                    self.explored_states.append(new_state)
```

- Makes board = (object) Boards [self. start, previous_board=none, direction=none, cost=0, heuristic=0]

## The A* Class contains the following:

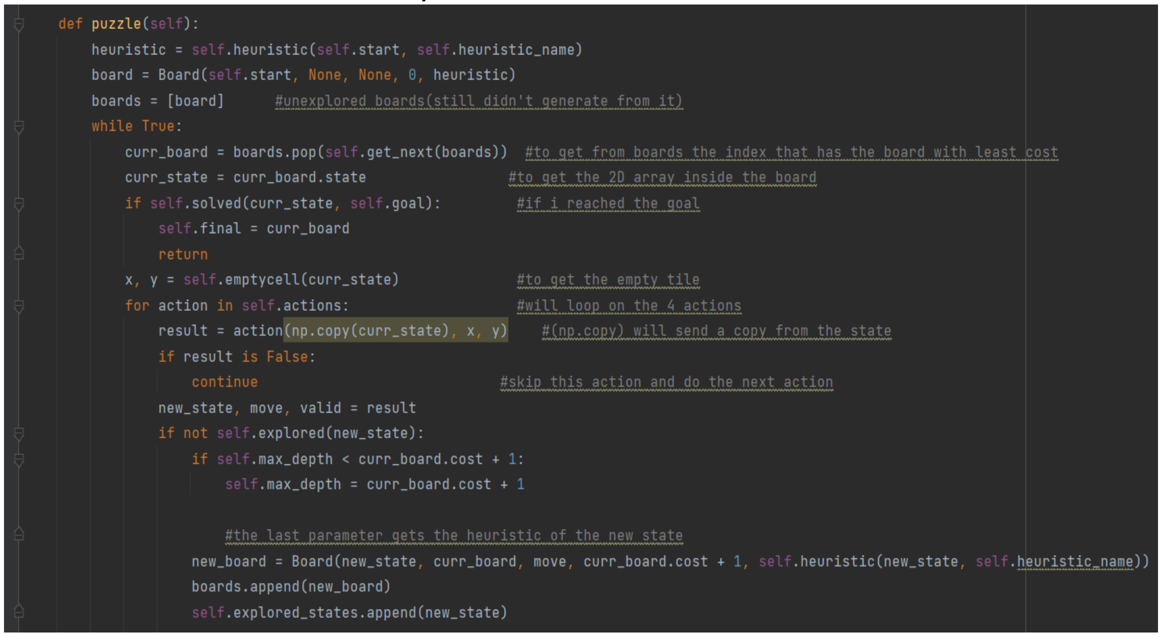
For implementation will use heap. The A* Minimize the total estimated solution cost.

1. The A* class will inherit all the functions from the Super class which is (The Puzzle class). Also, it is the same code as the BFS but with some differences.
    - First difference that we added and initialized the heuristic.



```python
from Board import Board
import numpy as np
from Puzzle import Puzzle

#inherit everything from the puzzle class
class ASTAR(Puzzle):
    # initialization
    def __init__(self, start, goal, heuristic_name):
        super().__init__(start, goal)
        self.final = None     #we initialize it with none because we still didn't reach to the goal
        self.path = []        #the array will be filled when we reach to the goal
        self.actions = [self.up, self.down, self.right, self.left]     #array from functions to loop on each action
        self.heuristic_name = heuristic_name
```

2. Define a function called puzzle



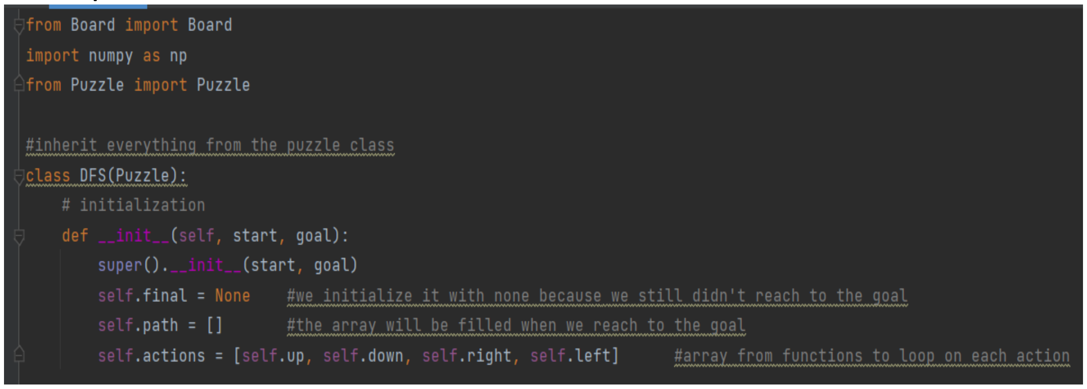
```python
    def puzzle(self):
        heuristic = self.heuristic(self.start, self.heuristic_name)
        board = Board(self.start, None, None, 0, heuristic)
        boards = [board]      #unexplored boards(still didn't generate from it)
        while True:
            curr_board = boards.pop(self.get_next(boards))  #to get from boards the index that has the board with least cost
            curr_state = curr_board.state                  #to get the 2D array inside the board
            if self.solved(curr_state, self.goal):          #if i reached the goal
                self.final = curr_board
                return
            x, y = self.emptycell(curr_state)               #to get the empty tile
            for action in self.actions:                     #will loop on the 4 actions
                result = action(np.copy(curr_state), x, y)    #(np.copy) will send a copy from the state
                if result is False:
                    continue                          #skip this action and do the next action
                new_state, move, valid = result
                if not self.explored(new_state):
                    if self.max_depth < curr_board.cost + 1:
                        self.max_depth = curr_board.cost + 1

                    #the last parameter gets the heuristic of the new state
                    new_board = Board(new_state, curr_board, move, curr_board.cost + 1, self.heuristic(new_state, self.heuristic_name))
                    boards.append(new_board)
                    self.explored_states.append(new_state)
```

- Second difference that in Makes board = (object) Boards [self. start, previous_board=none, direction=none, cost=0, heuristic]
  We added the heuristic and didn't initialize it with zero.

## **The DFS Class contains the following:**

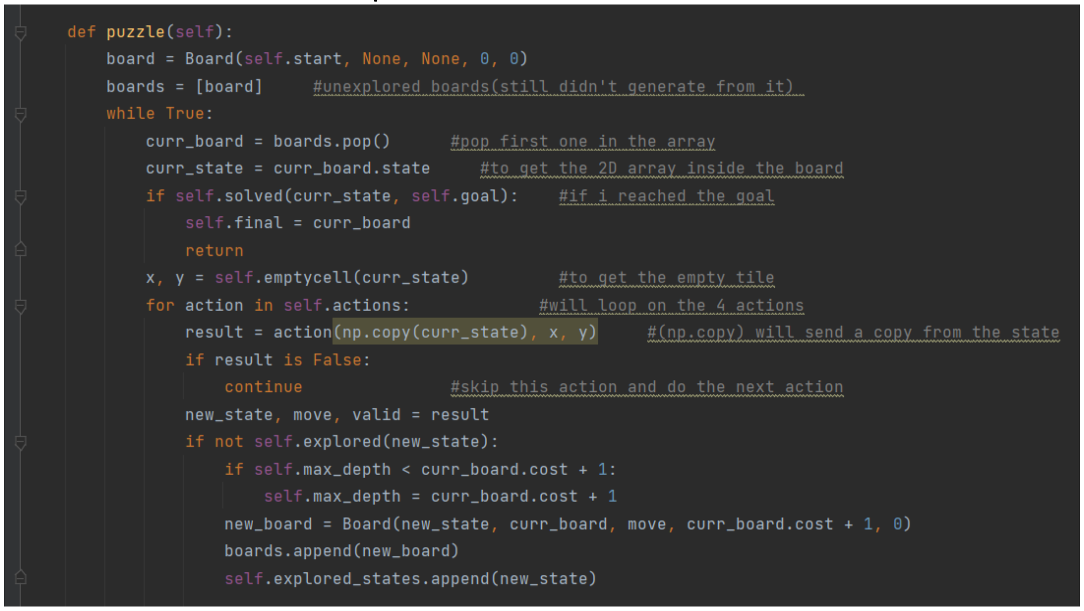
For implementation will use heap. The DFS expand deepest first.

1. The DFS class will inherit all the functions from the Super class which is (The Puzzle class). Also, it is the same code as the BFS exactly but the BFS with queue and the DFS with stack.



```
from Board import Board
import numpy as np
from Puzzle import Puzzle

#inherit everything from the puzzle class
class DFS(Puzzle):
    # initialization
    def __init__(self, start, goal):
        super().__init__(start, goal)
        self.final = None    #we initialize it with none because we still didn't reach to the goal
        self.path = []       #the array will be filled when we reach to the goal
        self.actions = [self.up, self.down, self.right, self.left]      #array from functions to loop on each action
```

2. Define a function called puzzle

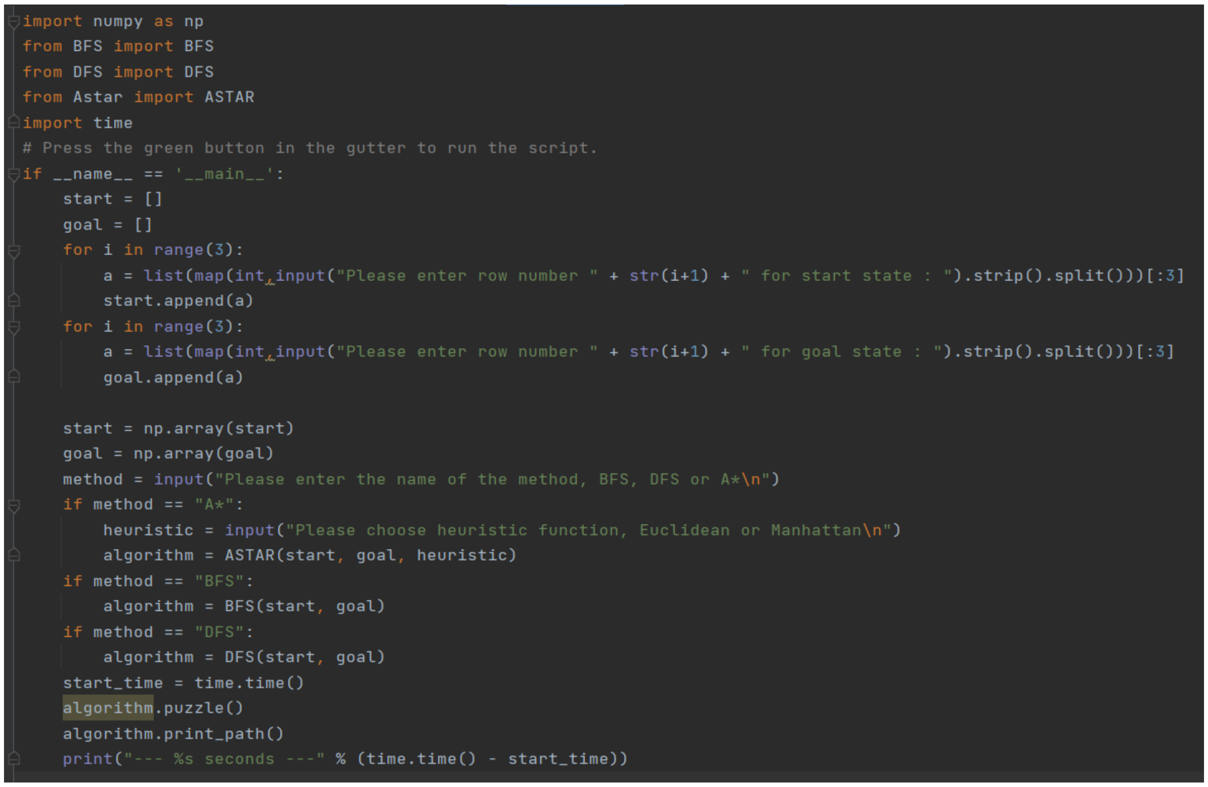


```
    def puzzle(self):
        board = Board(self.start, None, None, 0, 0)
        boards = [board]     #unexplored boards(still didn't generate from it)
        while True:
            curr_board = boards.pop()      #pop first one in the array
            curr_state = curr_board.state      #to get the 2D array inside the board
            if self.solved(curr_state, self.goal):    #if i reached the goal
                self.final = curr_board
                return
            x, y = self.emptycell(curr_state)          #to get the empty tile
            for action in self.actions:              #will loop on the 4 actions
                result = action(np.copy(curr_state), x, y)      #(np.copy) will send a copy from the state
                if result is False:
                    continue               #skip this action and do the next action
                new_state, move, valid = result
                if not self.explored(new_state):
                    if self.max_depth < curr_board.cost + 1:
                        self.max_depth = curr_board.cost + 1
                    new_board = Board(new_state, curr_board, move, curr_board.cost + 1, 0)
                    boards.append(new_board)
                    self.explored_states.append(new_state)
```

- Makes board = (object) Boards [self. start, previous_board=none, direction=none, cost=0, heuristic=0]

## The Main contains the following:

It calls the 3 search algorithms and asks the user to input the initial state and goal state. Also, the program asks the user which algorithm the user wants to search with and if he chooses A* the program will ask him whether he wants to continue with (Euclidean or Manhattan).

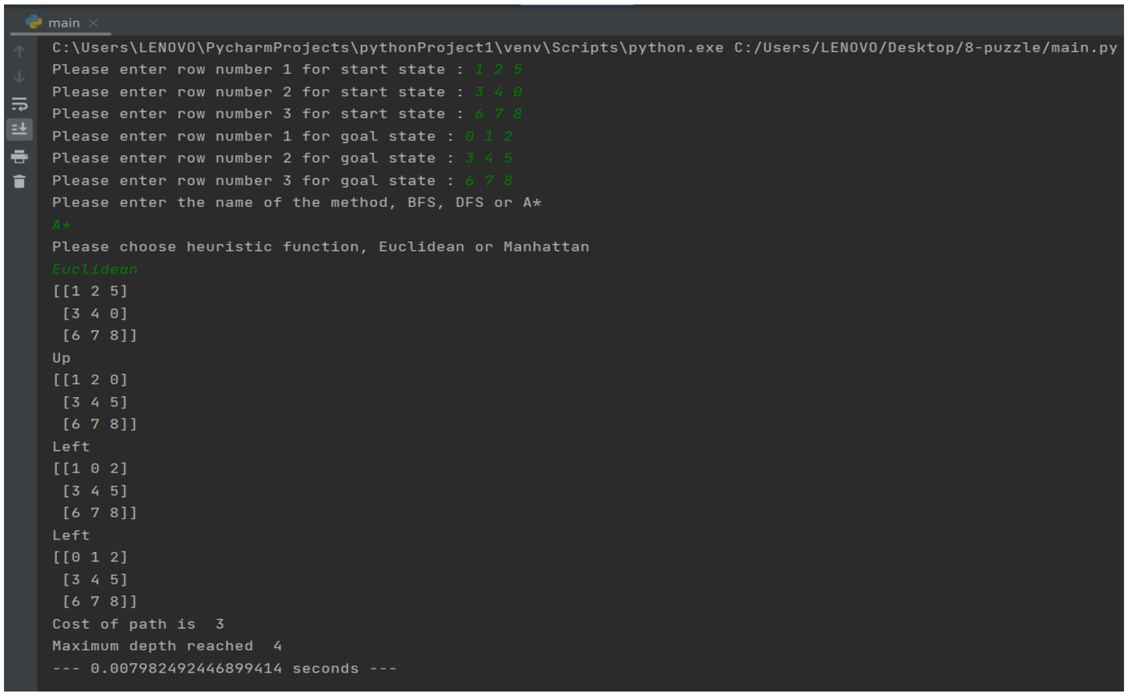


```python
import numpy as np
from BFS import BFS
from DFS import DFS
from Astar import ASTAR
import time
# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    start = []
    goal = []
    for i in range(3):
        a = list(map(int,input("Please enter row number " + str(i+1) + " for start state : ").strip().split()))[:3]
        start.append(a)
    for i in range(3):
        a = list(map(int,input("Please enter row number " + str(i+1) + " for goal state : ").strip().split()))[:3]
        goal.append(a)

    start = np.array(start)
    goal = np.array(goal)
    method = input("Please enter the name of the method, BFS, DFS or A*\n")
    if method == "A*":
        heuristic = input("Please choose heuristic function, Euclidean or Manhattan\n")
        algorithm = ASTAR(start, goal, heuristic)
    if method == "BFS":
        algorithm = BFS(start, goal)
    if method == "DFS":
        algorithm = DFS(start, goal)
    start_time = time.time()
    algorithm.puzzle()
    algorithm.print_path()
    print("--- %s seconds ---" % (time.time() - start_time))
```

Our program shows:

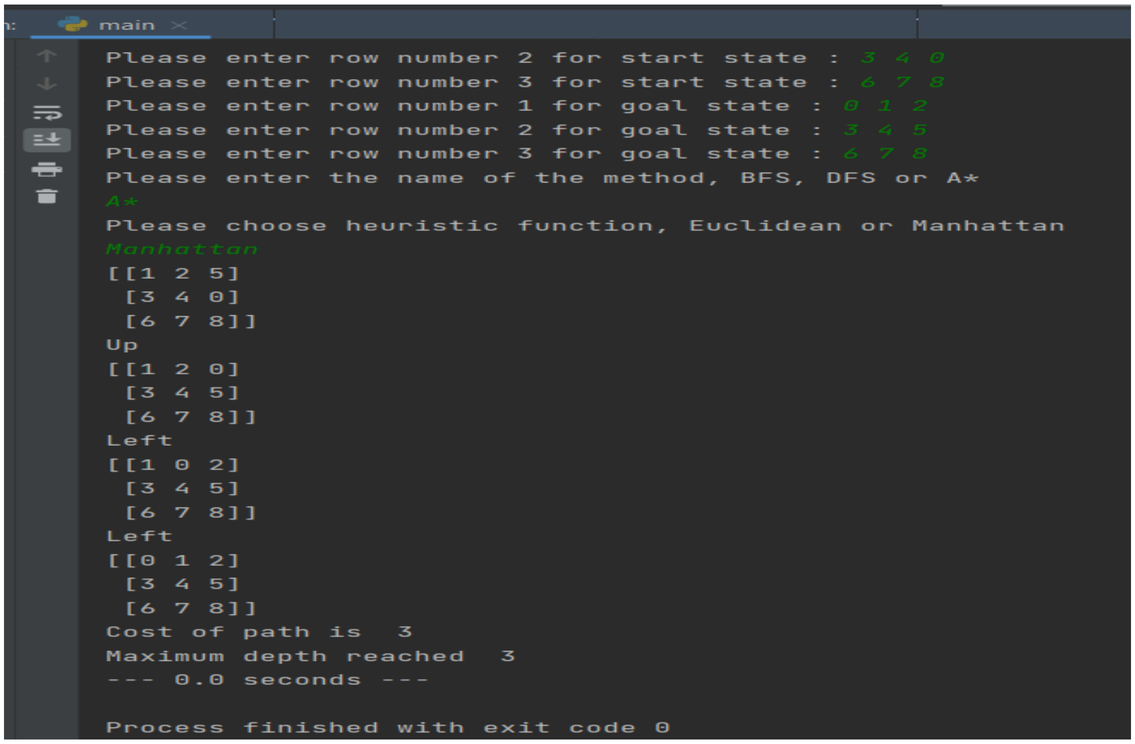
- Path to goal
- Cost of path
- Maximum depth
- Running time

Know we will show you an example with initial state = [[1,2,5], [3,4,0], [6,7,8]] and goal state = [[0,1,2], [3,4,5], [6,7,8]] with the 3 search algorithms. First with **A* (Euclidean)**



```
C:\Users\LENOVO\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:/Users/LENOVO/Desktop/8-puzzle/main.py
Please enter row number 1 for start state : 1 2 5
Please enter row number 2 for start state : 3 4 0
Please enter row number 3 for start state : 6 7 8
Please enter row number 1 for goal state : 0 1 2
Please enter row number 2 for goal state : 3 4 5
Please enter row number 3 for goal state : 6 7 8
Please enter the name of the method, BFS, DFS or A*
A*
Please choose heuristic function, Euclidean or Manhattan
Euclidean
[[1 2 5]
 [3 4 0]
 [6 7 8]]
Up
[[1 2 0]
 [3 4 5]
 [6 7 8]]
Left
[[1 0 2]
 [3 4 5]
 [6 7 8]]
Left
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Cost of path is  3
Maximum depth reached  4
--- 0.007982492446899414 seconds ---
```

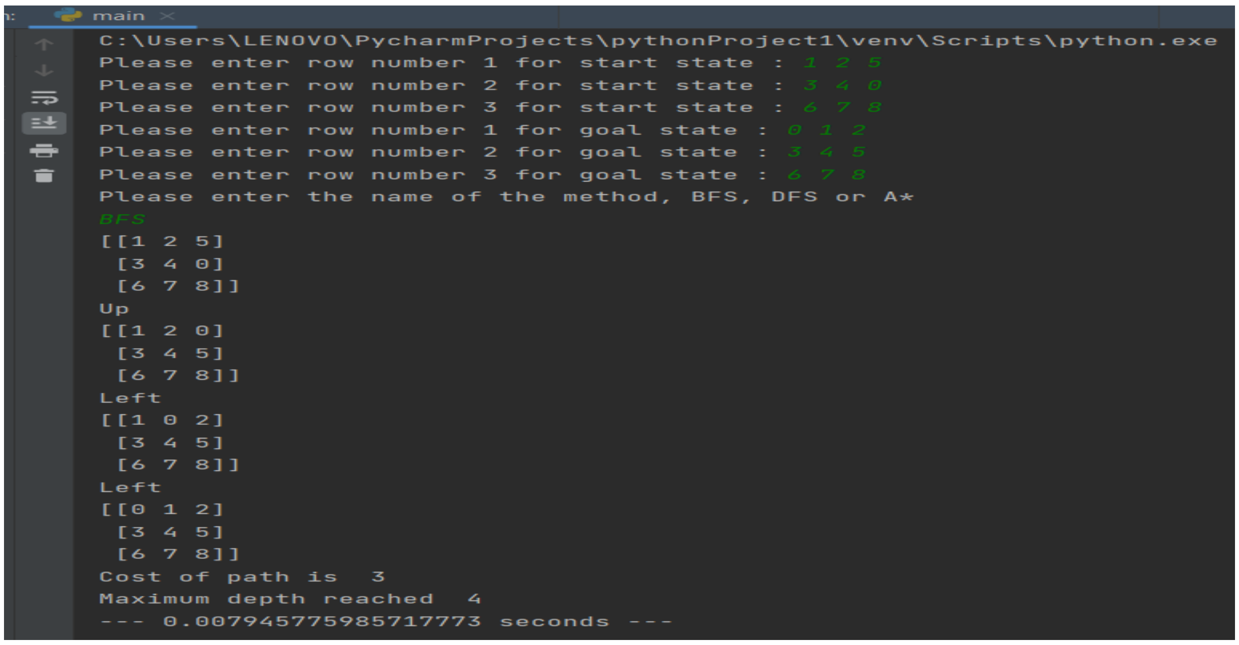
## A* (Manhattan)



```
Please enter row number 2 for start state : 3 4 0
Please enter row number 3 for start state : 6 7 8
Please enter row number 1 for goal state : 0 1 2
Please enter row number 2 for goal state : 3 4 5
Please enter row number 3 for goal state : 6 7 8
Please enter the name of the method, BFS, DFS or A*
A*
Please choose heuristic function, Euclidean or Manhattan
Manhattan
[[1 2 5]
 [3 4 0]
 [6 7 8]]
Up
[[1 2 0]
 [3 4 5]
 [6 7 8]]
Left
[[1 0 2]
 [3 4 5]
 [6 7 8]]
Left
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Cost of path is  3
Maximum depth reached  3
--- 0.0 seconds ---

Process finished with exit code 0
```

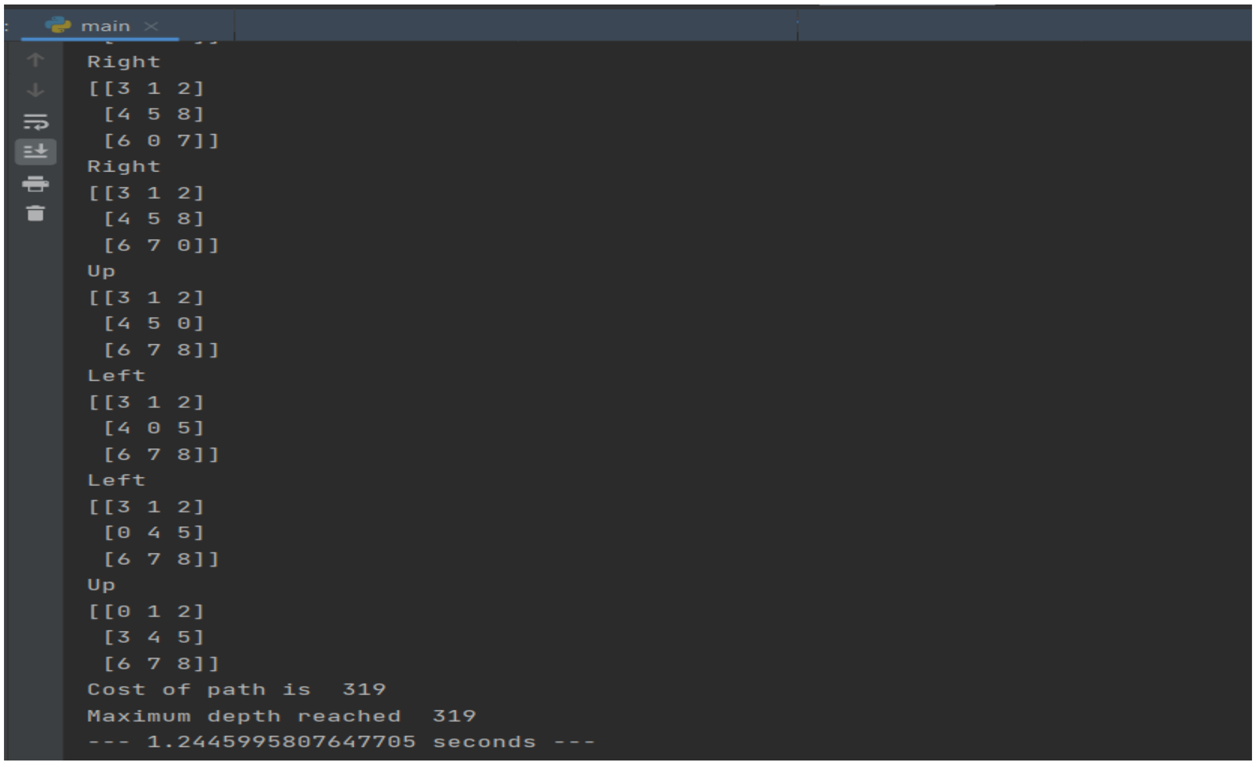
- We concluded that with Manhattan took least time and depth.

**BFS**



```
C:\Users\LENOVO\PycharmProjects\pythonProject1\venv\Scripts\python.exe
Please enter row number 1 for start state : 1 2 5
Please enter row number 2 for start state : 3 4 0
Please enter row number 3 for start state : 6 7 8
Please enter row number 1 for goal state : 0 1 2
Please enter row number 2 for goal state : 3 4 5
Please enter row number 3 for goal state : 6 7 8
Please enter the name of the method, BFS, DFS or A*
BFS
[[1 2 5]
 [3 4 0]
 [6 7 8]]
Up
[[1 2 0]
 [3 4 5]
 [6 7 8]]
Left
[[1 0 2]
 [3 4 5]
 [6 7 8]]
Left
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Cost of path is  3
Maximum depth reached  4
--- 0.007945775985717773 seconds ---
```

**DFS**



```
Right
[[3 1 2]
 [4 5 8]
 [6 0 7]]
Right
[[3 1 2]
 [4 5 8]
 [6 7 0]]
Up
[[3 1 2]
 [4 5 0]
 [6 7 8]]
Left
[[3 1 2]
 [4 0 5]
 [6 7 8]]
Left
[[3 1 2]
 [0 4 5]
 [6 7 8]]
Up
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Cost of path is  319
Maximum depth reached  319
--- 1.2445995807647705 seconds ---
```

- We concluded that BFS is optimal because we reached to the goal with a least cost but for DFS it's not optimal because we didn't reach the goal with the least cost, and it took larger time to reach to the goal.

GOALS
TEAMWORK
SUCCESS
SKILLS
VISION