Assignment 6

**Sarthak Kapaliya**

**20BCP072**

**G2**

# A\* is generally more efficient than Hill climbing in finding the optimal solution to the n-puzzle problem. However, Hill climbing can be faster in some cases and requires less memory to store the search tree.

import heapq

class Node:

    def \_\_init\_\_(self, state, parent=None, g=0, h=0):

        self.state = state

        self.parent = parent

        self.g = g

        self.h = h

    def f(self):

        return self.g + self.h

    def \_\_lt\_\_(self, other):

        return self.f() < other.f()

    def \_\_eq\_\_(self, other):

        return self.state == other.state

def misplaced\_tiles(state):

    goal\_state = [1, 2, 3, 8,0,4,7,6,5]

    return sum([1 for i in range(9) if state[i] != goal\_state[i]])

def get\_children(node):

    children = []

    blank\_index = node.state.index(0)

    if blank\_index % 3 > 0:

        left\_child\_state = node.state[:]

        left\_child\_state[blank\_index], left\_child\_state[blank\_index - 1] = left\_child\_state[blank\_index - 1], left\_child\_state[blank\_index]

        children.append(Node(left\_child\_state, node, node.g + 1, misplaced\_tiles(left\_child\_state)))

    if blank\_index % 3 < 2:

        right\_child\_state = node.state[:]

        right\_child\_state[blank\_index], right\_child\_state[blank\_index + 1] = right\_child\_state[blank\_index + 1], right\_child\_state[blank\_index]

        children.append(Node(right\_child\_state, node, node.g + 1, misplaced\_tiles(right\_child\_state)))

    if blank\_index // 3 > 0:

        up\_child\_state = node.state[:]

        up\_child\_state[blank\_index], up\_child\_state[blank\_index - 3] = up\_child\_state[blank\_index - 3], up\_child\_state[blank\_index]

        children.append(Node(up\_child\_state, node, node.g + 1, misplaced\_tiles(up\_child\_state)))

    if blank\_index // 3 < 2:

        down\_child\_state = node.state[:]

        down\_child\_state[blank\_index], down\_child\_state[blank\_index + 3] = down\_child\_state[blank\_index + 3], down\_child\_state[blank\_index]

        children.append(Node(down\_child\_state, node, node.g + 1, misplaced\_tiles(down\_child\_state)))

    return children

def a\_star(initial\_state):

    start\_node = Node(initial\_state, None, 0, misplaced\_tiles(initial\_state))

    open\_list = [start\_node]

    closed\_list = []

    while open\_list:

        current\_node = heapq.heappop(open\_list)

        if current\_node.h == 0:

            path = []

            while current\_node:

                path.append(current\_node.state)

                current\_node = current\_node.parent

            return list(reversed(path))

        closed\_list.append(current\_node)

        for child in get\_children(current\_node):

            if child in closed\_list:

                continue

            if child not in open\_list:

                heapq.heappush(open\_list, child)

            else:

                existing\_node = open\_list[open\_list.index(child)]

                if child.g < existing\_node.g:

                    existing\_node.g = child.g

                    existing\_node.parent = child.parent

    return None

if \_\_name\_\_ == "\_\_main\_\_":

    initial\_state = [2,8,3,1,6,4,7,0,5]

    path = a\_star(initial\_state)

    for state in path:

        print(state[0:3]) # print the first three elements

        print(state[3:6]) # print the next three elements

        print(state[6:9])

        print()

Output:

