I. CHECK LIST

UI console	**Done**
Define class of puzzle	**Done**
Define UCS search	**Done**
Define A* search	**Done**
Run 8 puzzle	**Done**
Run 15 puzzle	**Done**
Run 35 puzzle	**Done**

II. SET UP

1. Express each state of puzzle:

```
class Puzzle:
    def __init__(self, state, position_0, parent_cost, path_cost=0,
move=None, parent=None):
        self.state = state
        self.path_cost = path_cost + parent_cost
        self.move = move
        self.position_0 = position_0
        self.parent = parent
```

- ***state:*** This is the state of puzzle described by list of string from left to right, top to bottom. For example with 35-puzzle: ['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17','18','19','20','21','22','23','24','25','26','27','28','29','30','31','32','33','34','35','0'].
- ***position_0:*** This is the position of blank in the puzzle. In above example, the position of blank is at 35 (calculate from 0 in list).
- ***parent_cost:*** This is the total path cost from the start to the parent of the current puzzle.
- ***path_cost:*** This is the path cost from the parent to the current puzzle.
- ***move:*** This is the action of the blank from the parent to the current puzzle.
- ***parent:*** This is the parent of current puzzle.

2. Get all the neighbors of the current puzzle:

```
def child_puzzle(self, n, parent):

    child_expand = []

    if parent.move is None:
        self.move_up(n, self.path_cost, 1, parent, child_expand)
        self.move_down(n, self.path_cost, 1, parent, child_expand)
        self.move_left(n, self.path_cost, 1, parent, child_expand)
        self.move_right(n, self.path_cost, 1, parent, child_expand)
    elif parent.move == "up":
        self.move_up(n, self.path_cost, 1, parent, child_expand)
        self.move_left(n, self.path_cost, 1, parent, child_expand)
        self.move_right(n, self.path_cost, 1, parent, child_expand)
    elif parent.move == "down":
        self.move_left(n, self.path_cost, 1, parent, child_expand)
```

```
        self.move_right(n, self.path_cost, 1, parent, child_expand)
        self.move_down(n, self.path_cost, 1, parent, child_expand)
    elif parent.move == "left":
        self.move_left(n, self.path_cost, 1, parent, child_expand)
        self.move_up(n, self.path_cost, 1, parent, child_expand)
        self.move_down(n, self.path_cost, 1, parent, child_expand)
    elif parent.move == "right":
        self.move_right(n, self.path_cost, 1, parent, child_expand)
        self.move_down(n, self.path_cost, 1, parent, child_expand)
        self.move_up(n, self.path_cost, 1, parent, child_expand)

    return child_expand
```

- I assume that each step will cost 1.
- Consider the parent.move to ignore the return from the current puzzle to parent puzzle.

```
def move_up(self, n, parent_cost, path_cost, parent, child_expand):
    i = parent.position_0 // n
    j = parent.position_0 % n
    if i <= 0:
        return None
    up_state = parent.state.copy()
    up_state[(i - 1) * n + j], up_state[i * n + j] = up_state[i * n +
j], up_state[(i - 1) * n + j]
    temp = Puzzle(up_state, (i - 1) * n + j, parent_cost, path_cost,
"up", parent)
    child_expand.append(temp)

def move_down(self, n, parent_cost, path_cost, parent, child_expand):
    i = parent.position_0 // n
    j = parent.position_0 % n
    if i >= n - 1:
        return None
    down_state = parent.state.copy()
    down_state[(i + 1) * n + j], down_state[i * n + j] = down_state[i *
n + j], down_state[(i + 1) * n + j]
    temp = Puzzle(down_state, (i + 1) * n + j, parent_cost, path_cost,
"down", parent)
    child_expand.append(temp)

def move_left(self, n, parent_cost, path_cost, parent, child_expand):
    i = parent.position_0 // n
    j = parent.position_0 % n
    if j <= 0:
        return None
    left_state = parent.state.copy()
    left_state[i * n + j], left_state[i * n + j - 1] = left_state[i * n
+ j - 1], left_state[i * n + j]
    temp = Puzzle(left_state, i * n + j - 1, parent_cost, path_cost,
"left", parent)
    child_expand.append(temp)

def move_right(self, n, parent_cost, path_cost, parent, child_expand):
    i = parent.position_0 // n
    j = parent.position_0 % n
    if j >= n - 1:
        return None
    right_state = parent.state.copy()
```

```
    right_state[i * n + j], right_state[i * n + j + 1] = right_state[i
* n + j + 1], right_state[i * n + j]
    temp = Puzzle(right_state, i * n + j + 1, parent_cost, path_cost,
"right", parent)
    child_expand.append(temp)
```

- For each move of the blank from the current puzzle, we consider the position of blank. If it is at left corner, we don't return the left action for the blank of current puzzle. This is the same for right, up, down action.

3. Calculate the heuristic:

```
def heuristic(self, n):
    state = [int(x) for x in self.state]
    vertical = sum(1 for i in range(len(state)) if state[i] != 0 for j
in range(i + 1, len(state)) if
                   state[i] > state[j] and state[j] != 0)
    horizon_string = [state[j] for i in range(n) for j in range(i,
len(state), n)]
    horizontal = sum(
        abs(i - goalstate2[int(horizon_string[i])]) for i in
range(len(horizon_string)) if horizon_string[i] != 0)
    return vertical // 3 + vertical % 3 + horizontal // 3 + horizontal
% 3
```

- ***goalstate2:*** this is the global variable described by the dict type in python (key: the tile, value: position of this tile in the list) to calculate when considering the horizontal moves. This action will save a lot of time when executing the heuristic function in A* search.
    - With 8-puzzle `goalstate2 = {1:0,4:1,7:2,2:3,5:4,8:5,3:6,6:7,0:8}`
    - With 15-puzzle `goalstate2 = {1:0, 5:1,9:2,13:3,2:4,6:5,10:6,14:7,3:8,7:9,11:10,15:11,4:12,8:13,12:14,0:15}`
    - With 35-puzzle `goalstate2 = {1:0,7:1,13:2,19:3,25:4,31:5,2:6,8:7,14:8,20:9,26:10,32:11,3:12,9:13,15:14,21:15,27:16,33:17,4:18,10:19,16:20,22:21,28:22,34:23,5:24,11:25,17:26,23:27,29:28,35:29,6:30,12:31,18:32,24:33,30:34,0:35}`
- By default, the goal state is the ascending list with the blank at the final.
- The way to calculate heuristic is presented by Michial Kim at Michael Kim | Solving the 15 Puzzle.

4. UCS search:

```
5. def UCS(initial, goal, n):
       frontier = PriorityQueue(0)
       explored: dict[str, int] = {}
       open_list: dict[str, int] = {}
```

```
    frontier.put([0, initial])

    while True:
        if frontier.qsize() == 0:
            return explored, "failure"
        current_puzzle = frontier.get()[-1]
        current_state = ''.join(current_puzzle.state)
        if explored.get(current_state) == 1:
            continue
        if current_puzzle.state == goal:
            return explored, "success"
        explored[current_state] = 1

        for child in current_puzzle.child_puzzle(n, current_puzzle):
            temp = ''.join(child.state)
            if explored.get(temp) is None:
                if open_list.get(temp) is None or open_list.get(temp)
>= child.path_cost:
                    frontier.put([child.path_cost, child])
                    open_list[temp] = child.path_cost
```

- This function is built based on pseudo-code:

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the element
  explored ← an empty set
  loop do
    if EMPTY?( frontier) then return failure
    node ← POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored and not in frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
                                                                        25
```

- ***goal***: This is the state of goal described by the list of ascending string with the blank at the final.
- ***n*:** This is the size of the puzzle. For example, with the 8-puzzle: n is 3, with the 15-puzzle: n is 4…
- Moreover, I add open_list to compare the higher PATH-COST when considering all the neighbors of current puzzle. Because the open_list is a dict variable, it will run faster than enumeration by for loop to all the queue in frontier.

6. A* search

```python
def AstartSearch(initial, goal, n):
    frontier = PriorityQueue(0)
    explored: dict[str, int] = {}
    open_list: dict[str, int] = {}

    frontier.put([initial.heuristic(n) + initial.path_cost, initial])

    while True:
        if frontier.qsize() == 0:
            return explored, "failure"
        current_puzzle = frontier.get()[-1]
        current_state = ''.join(current_puzzle.state)
        if explored.get(current_state) == 1:
            continue
        if current_puzzle.state == goal:
            return explored, "success"
        explored[current_state] = 1

        for child in current_puzzle.child_puzzle(n, current_puzzle):
            temp = ''.join(child.state)
            if explored.get(temp) is None:
                heuristic = child.heuristic(n)
                if open_list.get(temp) is None or open_list.get(temp)
>= (child.path_cost + heuristic):
                    frontier.put([child.path_cost + heuristic, child])
                    open_list[temp] = child.path_cost + heuristic

        del current_puzzle
```

- The algorithm is quite similar to UCS. However, it uses heuristic to decide the next state.

7. **Modification for path:**

- I modify A* search and UCS to print path from start to goal if they find the path.

```python
-   def AstartSearch(initial, goal, n):
    frontier = PriorityQueue(0)
    explored: dict[str, int] = {}
    open_list: dict[str, int] = {}

    frontier.put([initial.heuristic(n) + initial.path_cost, initial])

    while True:
        if frontier.qsize() == 0:
            return explored, "failure", None
        current_puzzle = frontier.get()[-1]
        current_state = ''.join(current_puzzle.state)
        if explored.get(current_state) == 1:
            continue
        if current_puzzle.state == goal:
            path_list = []
            while current_puzzle is not None:
                path_list.append(current_puzzle)
                current_puzzle = current_puzzle.parent
            path_list.reverse()
            return explored, "success", path_list
        explored[current_state] = 1
```

```
        for child in current_puzzle.child_puzzle(n, current_puzzle):
            temp = ''.join(child.state)
            if explored.get(temp) is None:
                heuristic = child.heuristic(n)
                if open_list.get(temp) is None or open_list.get(temp)
>= (child.path_cost + heuristic):
                    frontier.put([child.path_cost + heuristic, child])
                    open_list[temp] = child.path_cost + heuristic

        del current_puzzle


def UCS(initial, goal, n):
    frontier = PriorityQueue(0)
    explored: dict[str, int] = {}
    open_list: dict[str, int] = {}

    frontier.put([0, initial])

    while True:
        if frontier.qsize() == 0:
            return explored, "failure", None
        current_puzzle = frontier.get()[-1]
        current_state = ''.join(current_puzzle.state)
        if explored.get(current_state) == 1:
            continue
        if current_puzzle.state == goal:
            path_list = []
            while current_puzzle is not None:
                path_list.append(current_puzzle)
                current_puzzle = current_puzzle.parent
            path_list.reverse()
            return explored, "success", path_list
        explored[current_state] = 1

        for child in current_puzzle.child_puzzle(n, current_puzzle):
            temp = ''.join(child.state)
            if explored.get(temp) is None:
                if open_list.get(temp) is None or open_list.get(temp)
>= child.path_cost:
                    frontier.put([child.path_cost, child])
                    open_list[temp] = child.path_cost
```

- I also write the printPath function to print this path:

```
def printPath(c,n):
    for t in c:
        j = 0
        print(t.path_cost)
        for i in t.state:
            if j % n == n-1:

                print(i, end=' ')
                print()
            else:
                print(i, end=' ')
```

```
            j += 1
        print("--------------------")
```

- c is the path_list or None returned by the search function.

III. 8-PUZZLE

1. Input:

1	2	3
	4	6
7	5	8
Start		

1	2	3
4	5	6
7	8	
Goal		

2. Trials:

Algorithm	Running time (milliseconds)	Memory (MB)
UCS	0.628	0.009
A*	0.735	0.004

3. Comment:

- When running 8-puzzle, both algorithms find the path from the start state to the goal state. Among them, UCS runs faster than A* but has more consumed memory than A*.

IV. 15-PUZZLE

1. Input:

8		6	3
14	15	10	7
2	9	5	13
12	1	4	11
Start			

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	
Goal			

2. Trials:

Algorithm	Running time (milliseconds)	Memory (MB)
UCS (Intractable)	1315000	exceeding memory (RAM: max 30GB)
A*	2862.916	15.31

3. **Comment:**

- When running 15-puzzle, UCS cannot find the solutions and fail after 1315 (seconds) because of the lack of memory. While A* can find the solutions after 4182.506 (milliseconds) and cost about 15.31 MB.

V. 35-PUZZLE

1. **Input:**

10	**35**	**18**	**12**	**27**	**31**
25	**16**	**32**	**2**	**28**	**7**
1	**24**	**5**	**20**	**26**	**34**
19	**17**	**3**	**9**	**14**	**8**
33	**22**	**13**	**15**	**4**	**6**
29	**21**	**11**		**23**	**30**
Start					

1	**2**	**3**	**4**	**5**	**6**
7	**8**	**9**	**10**	**11**	**12**
13	**14**	**15**	**16**	**17**	**18**
19	**20**	**21**	**22**	**23**	**24**
25	**26**	**27**	**28**	**29**	**30**
31	**32**	**33**	**34**	**35**	
Goal					

2. **Trials:**

Algorithm	**Running time (milliseconds)**	**Memory (MB)**
UCS (Intractable)	964000	exceeding memory (RAM: max 30GB)
A* (Intractable)	5219500	exceeding memory (RAM: max 30GB)

3. **Comment:**

- When running 35-puzzle, both algorithms fail to find the solution. While UCS fails after 964000 (seconds) because of lack of memory, A* fails after 5219500 (seconds) with the same reason.

VI. Conclusion

- A* might take more time than UCS because it must recalculate the heuristic at each step when considering neighbors, but not much in the case of the 8-puzzle. However, for UCS, the cost for all path-costs is the same, so it will store many states, leading to a quick overflow of memory. While A* optimizes the selection, it will reduce the overflow of memory. In the 15-puzzle, A* finds the answer, but UCS does not.

VII. Explain main function:

```
VIII. if __name__ == '__main__':

        while True:
            n = input("Please enter n with n*n is the size of puzzle or
    0 to exit: \n")
```

```
        if n == '0':
            break
        elif n.isdecimal():
            n = int(n)
            if n == 3 or n == 4 or n == 6:
                npuzzle = input("Please enter the puzzle for
example(0,1,2,3,4,5,6,7,8) or only 1 to run default: \n")
                initial = None
                goal = None
                if n == 3:
                    goalstate2 = {1: 0, 4: 1, 7: 2, 2: 3, 5: 4, 8:
5, 3: 6, 6: 7, 0: 8}
                    if npuzzle == '1':
                        initial = Puzzle(['1', '2', '3', '0', '4',
'6', '7', '5', '8'], 3, 0, 0)
                    else:
                        puzzle = npuzzle.split(',')
                        position_0 = puzzle.index('0')
                        initial = Puzzle(puzzle, position_0, 0, 0)
                    goal = ['1', '2', '3', '4', '5', '6', '7', '8',
'0']
                if n == 4:
                    goalstate2 = {1: 0, 5: 1, 9: 2, 13: 3, 2: 4, 6:
5, 10: 6, 14: 7, 3: 8, 7: 9, 11: 10, 15: 11, 4: 12,
                                  8: 13, 12: 14, 0: 15}
                    if npuzzle == '1':
                        initial = Puzzle(
                            ['8', '0', '6', '3', '14', '15', '10',
'7', '2', '9', '5', '13', '12', '1', '4', '11'], 1,
                            0, 0)
                    else:
                        puzzle = npuzzle.split(',')
                        position_0 = puzzle.index('0')
                        initial = Puzzle(puzzle, position_0, 0, 0)
                    goal = ['1', '2', '3', '4', '5', '6', '7', '8',
'9', '10', '11', '12', '13', '14', '15', '0']
                if n == 6:
                    goalstate2 = {1: 0, 7: 1, 13: 2, 19: 3, 25: 4,
31: 5, 2: 6, 8: 7, 14: 8, 20: 9, 26: 10, 32: 11,
                                  3: 12, 9: 13, 15: 14, 21: 15, 27:
16, 33: 17, 4: 18, 10: 19, 16: 20, 22: 21, 28: 22,
                                  34: 23, 5: 24, 11: 25, 17: 26, 23:
27, 29: 28, 35: 29, 6: 30, 12: 31, 18: 32, 24: 33,
                                  30: 34, 0: 35}
                    if npuzzle == '1':
                        initial = Puzzle(
                            ['10', '35', '18', '12', '27', '31',
'25', '16', '32', '2', '28', '7', '1', '24', '5', '20',
                             '26', '34', '19', '17', '3', '9', '14',
'8', '33', '22', '13', '15', '4', '6', '29', '21',
                             '11', '0', '23', '30'], 33, 0, 0)
                    else:
                        puzzle = npuzzle.split(',')
                        position_0 = puzzle.index('0')
                        initial = Puzzle(puzzle, position_0, 0, 0)
                    goal = ['1', '2', '3', '4', '5', '6', '7', '8',
'9', '10', '11', '12', '13', '14', '15', '16', '17',
```

```
                                 '18', '19', '20', '21', '22', '23',
'24', '25', '26', '27', '28', '29', '30', '31', '32',
                                 '33', '34', '35', '0']
                while True:
                    choose = input("Please choose search with UCS: 1
and A*: 2 or 0 to return: \n")
                    if choose == '0':
                        break
                    elif choose.isdecimal():
                        choose = int(choose)
                        if choose == 1:
                            print("UCS search")
                            time_start = datetime.datetime.now()
                            tracemalloc.start()
                            a, b, c = UCS(initial, goal, n)
                            current, peak =
tracemalloc.get_traced_memory()
                            time_end = datetime.datetime.now()
                            print(f'time: {(time_end -
time_start).total_seconds() * 1000} milliseconds')
                            print(f"memory now: {current / (1024 *
1024)} MB")
                            print(f"memory max: {peak / (1024 *
1024)} MB")
                            tracemalloc.stop()
                            print(b)
                            printPath(c, n)
                            print("----------------------------")
                        elif choose == 2:
                            print("A* search")
                            time_start = datetime.datetime.now()
                            tracemalloc.start()
                            a, b, c = AstartSearch(initial, goal, n)
                            current, peak =
tracemalloc.get_traced_memory()
                            time_end = datetime.datetime.now()
                            print(f'time: {(time_end -
time_start).total_seconds() * 1000} milliseconds')
                            print(f"memory now: {current / (1024 *
1024)} MB")
                            print(f"memory max: {peak / (1024 *
1024)} MB")
                            tracemalloc.stop()
                            print(b)
                            if c is not None:
                                printPath(c, n)
                            print("----------------------------")
                        else:
                            print("Please choose correctly")
            else:
                print("n is only 3,4 or 6")
```

- At first, user inputs n. The constraint here is that n is only equal to 3,4 or 6, and 0 to exit.
- Then user can input 1 to run default puzzle or input your puzzle with required format (Ex: 1,2,3,4,5,6,7,8,0 for 8-puzzle)

- Then, user inputs 1 or 2 to choose UCS or A* to run, and 0 to return the back.
- The results return the running the time and consumed memory.
- Furthermore, you can only change the initial with the structure mentioned above. The goal, goalstate2 must remain.