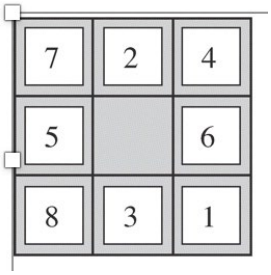
CPSC 4420/6420: ARTIFICIAL INTELLIGENCE

ASSIGNMENT 1- DUE: SEP 16, 2022 @11:59 PM

NAME: PARAMPREET SINGH



For the above puzzle shown here, develop a Python program that

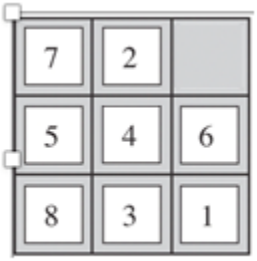
(A) [10 pts] Lists all states [No need to submit the output (the list of states) for this part, and just submit the code and the first [or randomly selected] 10 states, since the list will be very long !!!]

(B) [10 pts] Gets the current state and the action (moving up:1, down:2, left:3, right:4) as input, and returns the resulting state. Represent the blank spot with "0" and use one of the following naming formats for states

- Represent each state with a sequence of numbers from left to right and top to bottom. Ex. Use 7-2-4-5-0-6-8-3-1 for the state shown above
- Represent each state by a 9-digit integer number. Like show the above state by 724506831

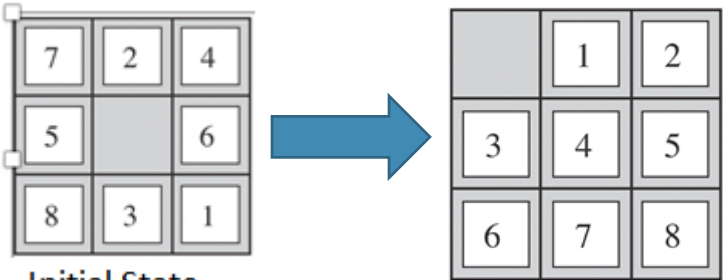
Ex: Input (Current state: 724506831, Action: 3) should give output state: 724056831

(C) [10 pts] Suppose that the goal is to arrange the numbers so that the resulting 3-digit numbers created by each row are divisible by 3. For instance, 7-2-0-5-4-6-8-3-1 is a goal state because 720, 546, and 831 are divisible by 3. Write a program that prompts the user to receive an arbitrary initial state, and then performs random actions to reach the goal state. Show the sequence of actions and the sequence of states.



Sample goal state

(D) [20 pts] Suppose that the goal is arranging the blocks in numerical order as shown below. Develop a Breadth First Search (BFS) algorithm and show the results. Present the sequence of states and moves, starting from the initial state. How many moves (actions) did it take to reach the goal state?

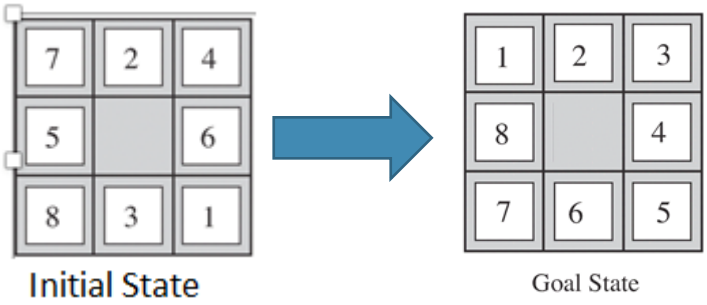


Initial State

Goal State

(E) [10 pts] Repeat part (D) using a Depth-First Search (DFS). How many moves (actions) did it take to reach the goal state?
Which algorithm found the solution with fewer moves? Explain your observation.

(F) [10 pts] Repeat Part (D), if the goal is ordering the numbers clockwise around the blank space, with the given initial state, as shown below.



(G) Implement a Uniform Cost Search (UCS), if the goal is achieving the final state in part F from an arbitrary initial state, if we have the following costs for different moves

[15 pts] G1) All moves have a unit cost
[15 pts] G2) Up (Cost=1), Down (Cost=1) Left (Cost=2) Right (Cost=0.5)
Present the sequences of moves and states for each option. How many actions are used to achieve the solution for each option? Explain your observation.

```python
# Answer A

# Importing 'permutations' function from "itertools" module to list a state space

from itertools import permutations

# used a while loop to ensure correct user input
user_input = ''
while len(user_input) != 9:
    user_input = list(input("Please enter the puzzle piece numbers(9 digit long)"))
    if len(user_input) != 9:
        print("Entered number is not 9 digit")

# input taken for number of states to print
print_states = int(input("Please enter the desired no of states you wish to print"))

# used permutations function from itertools module to store all states as a list
all_states = list(permutations(user_input))
print("State Space - ", len(all_states))

# print the desired number of states
for i in range(0, print_states):
    a, b, c, d, e, f, g, h, i = all_states[i]
    print(a+" "+b+" "+c)
    print(d+" "+e+" "+f)
    print(g+" "+h+" "+i + "\n")

```

```python
# Answer B

# taking the current state and action from user to return the output state

# defined variables
input_list = []
action_input = 0
blank_spot_idx = 0

# user input taken for puzzle piece sequence
while len(input_list) != 9:
    input_list = list(input("Please enter the puzzle piece numbers(9 digit long)"))
    if len(input_list) != 9:
        print("Entered number is not 9 digit")

# identified the index position of '0' in input state list
blank_spot_idx = input_list.index("0")

# action input taken from user while simultaneously filtered invalid entries
while action_input not in range(1, 5):
    action_input = int(input("Please enter the action for 'blank spot'\n Please note: \n moving up:1\n down:2 "
                             "\n left:3\n right:4 \n Action - "))
    if action_input not in range(1, 5):
        print("Invalid Entry! Please enter correct value")
    elif blank_spot_idx in [0, 1, 2]:
        if action_input == 1:
            print("Restricted Movement!!")
            action_input = 0
    elif blank_spot_idx in [0, 3, 6]:
        if action_input == 3:
            print("Restricted Movement!!")
            action_input = 0
    elif blank_spot_idx in [2, 5, 8]:
```

```
34          if action_input == 4:
35              print("Restricted Movement!!")
36              action_input = 0
37      elif blank_spot_idx in [6, 7, 8]:
38          if action_input == 2:
39              print("Restricted Movement!!")
40              action_input = 0
41
42  # action from current state to output state taken
43  if action_input == 1:
44      input_list[blank_spot_idx] = input_list[
    blank_spot_idx - 3]
45      input_list[blank_spot_idx - 3] = 0
46  elif action_input == 2:
47      input_list[blank_spot_idx] = input_list[
    blank_spot_idx + 3]
48      input_list[blank_spot_idx + 3] = 0
49  elif action_input == 3:
50      input_list[blank_spot_idx] = input_list[
    blank_spot_idx - 1]
51      input_list[blank_spot_idx - 1] = 0
52  else:
53      input_list[blank_spot_idx] = input_list[
    blank_spot_idx + 1]
54      input_list[blank_spot_idx + 1] = 0
55
56  # print output
57  print("Output State -\n" + f'{"".join(map(str,
    input_list))}')
58
```

```python
# Answer C

import random

# defined variables
user_input = ''
input_list = []
record = []
action_count = []
t = True
counter = 0

# index of directions list gives the possible actions on that index
directions = [[(1, 'right'), (3, 'down')],
              [(4, 'down'), (0, 'left'), (2, 'right')],
              [(5, 'down'), (1, 'left')],
              [(0, 'up'), (4, 'right'), (6, 'down')], [(1, 'up'), (5, 'right'), (7, 'down'), (3, 'left')],
              [(2, 'up'), (8, 'down'), (4, 'left')],
              [(3, 'left'), (7, 'right')],
              [(4, 'up'), (8, 'right'), (6, 'left')],
              [(5, 'up'), (7, 'left')]]

# user input taken for puzzle piece sequence
while len(user_input) != 9:
    user_input = input("Please enter the puzzle piece numbers(9 digit long)")
    if len(user_input) != 9:
        print("Entered number is not 9 digit")

# converted input to int list
input_list = [int(i) for i in user_input]

# initiated record for sequence of state, action
record.append((input_list, "initial"))

# loop to continue the game until goal state is achieved
```

```
36 while t:
37     if sum(input_list[0:3]) % 3 == 0 and sum(
   input_list[3:6]) % 3 == 0 and sum(input_list[6:9]) %
   3 == 0:
38         print("Input State is already Goal State!")
39         break
40 # logic to generate child node
41     idx = input_list.index(0)
42     i, action = random.choice(directions[idx])
43     child = input_list[:]
44     child[idx] = input_list[i]
45     child[i] = 0
46     input_list = child[:]
47     record.append((input_list, action))
48     counter += 1
49     # Check if child node is goal state then print
   the required data
50     if sum(input_list[0:3]) % 3 == 0 and sum(
   input_list[3:6]) % 3 == 0 and sum(input_list[6:9]) %
   3 == 0:
51         print("Goal State -\n", input_list)
52         print("Sequence of states -\n", record)
53         print("No of actions taken -\n", counter)
54         t = False
55         break
56
```

```python
# Answer D

# To reach goal state - 012345678 from initial state - 724506831 using BFS:
# No of actions taken - 173341

import queue

# defined variables

idx: int = 0
input_list = []
input_list_record = []
queue_child = queue.Queue()
goal_state = ['0', '1', '2', '3', '4', '5', '6', '7', '8']
record = []
counter = 0
child = []
record_print = []

# index of directions list gives the possible actions on that index
directions = [[1, 3], [4, 0, 2], [5, 1], [0, 4, 6], [1, 5, 7, 3], [2, 8, 4], [3, 7], [4, 8, 6], [5, 7]]

# if user input required for puzzle piece sequence
while len(input_list) != 9:
    input_list = list(input("Please enter the puzzle piece numbers(9 digit long)"))
    if len(input_list) != 9:
        print("Entered number is not 9 digit")

# initiated queue for BFS
queue_child.put(input_list)
# initiated sequence record of visited states
record.append(input_list)

# loop to continue the game until goal state is achieved
while not queue_child.empty():
```

```
36      if input_list == goal_state:
37          print("Input State is already Goal State!")
38          break
39      # logic to generate BFS child node
40      input_list = queue_child.get()
41      if input_list not in input_list_record:
42          input_list_record.append(input_list)
43          idx = input_list.index('0')
44          for i in directions[idx]:
45              child = input_list[:]
46              child[idx] = input_list[i]
47              child[i] = '0'
48              if child not in record:
49                  queue_child.put(child)
50                  record.append(child)
51                  record_print.append(''.join(map(str,
    child)))
52                  counter += 1
53                  print(counter)
54              # Check if child node is goal state then
    print the required data
55              if child == goal_state:
56                  print("Goal State -\n", child)
57                  print("Sequence -\n", record_print)
58                  print("No of actions taken -\n",
    counter)
59                  queue_child.queue.clear()
60                  break
61
62
63
```

```python
# Answer E

# No of actions taken -
# 172692

# defined variables
input_list = []
idx: int = 0
queue_list: list = []
goal_state = ['0', '1', '2', '3', '4', '5', '6', '7', '8']
record = []
record_temp = []
counter = 0

# index of directions list gives the possible actions on that index
directions = [[1, 3], [4, 0, 2], [5, 1], [0, 4, 6], [1, 5, 7, 3], [2, 8, 4], [3, 7], [4, 8, 6], [5, 7]]

# user input taken for puzzle piece sequence
while len(input_list) != 9:
    input_list = list(input("Please enter the puzzle piece numbers(9 digit long)"))
    if len(input_list) != 9:
        print("Entered number is not 9 digit")

# initiated queue for DFS
queue_list.append(input_list)

# loop to continue the game until goal state is achieved
while True:
    if input_list == goal_state:
        print("Input State is already Goal State!")
        break
    # logic to generate DFS child node
    input_list = queue_list.pop()
    if input_list not in record_temp:
        record_temp.append(input_list)
        idx = input_list.index('0')
```

```
37          for i in directions[idx]:
38              test = input_list[:]
39              test[idx] = test[i]
40              test[i] = '0'
41              queue_list.append(test)
42      # initiated sequence record of visited states
43          record.append("".join(map(str, input_list)))
44          counter += 1
45          # Check if child node is goal state then
    print the required data
46          if input_list == goal_state:
47              print("Sequence -\n" + f'{record}')
48              print("Goal State -\n" + f'{input_list}')
49              print("No of actions taken -\n" + f'{
    counter}')
50              break
51
```

```python
# Answer F

# For a given goal state, there are only 9!/2 initial states possible to reach that goal state.
# For the given initial state in part (F) - 724506831, the required goal state - 123804765 is not achievable
# Hence, some other arbitrary initial state may result in the required goal state,
# but the initial state given in question cannot yield the required goal state

import queue


# defined variables

idx: int = 0
input_list = []
input_list_record = []
queue_child = queue.Queue()
goal_state = []
record = []
counter = 0
child = []
record_print = []


# index of directions list gives the possible actions on that index
directions = [[1, 3], [4, 0, 2], [5, 1], [0, 4, 6], [1, 5, 7, 3], [2, 8, 4], [3, 7], [4, 8, 6], [5, 7]]

# user input taken for puzzle piece sequence
while len(input_list) != 9:
    input_list = list(input("Please enter the puzzle piece numbers(9 digit long)"))
    if len(input_list) != 9:
        print("Entered number is not 9 digit")
# user input taken for goal state sequence
while len(goal_state) != 9:
```

```
34      goal_state = list(input("Please enter the Goal
   State(9 digit long)"))
35      if len(goal_state) != 9:
36          print("Entered number is not 9 digit")
37
38  queue_child.put(input_list)
39  record.append(input_list)
40
41  while not queue_child.empty():
42      if input_list == goal_state:
43          print("Input State is already Goal State!")
44          break
45
46      input_list = queue_child.get()
47      if input_list not in input_list_record:
48          input_list_record.append(input_list)
49          idx = input_list.index('0')
50          for i in directions[idx]:
51              child = input_list[:]
52              child[idx] = input_list[i]
53              child[i] = '0'
54              if child not in record:
55                  queue_child.put(child)
56                  record.append(child)
57                  record_print.append(''.join(map(str,
   child)))
58                  counter += 1
59              if child == goal_state:
60                  print("Goal State -\n", child)
61                  print("Sequence -\n", record_print)
62                  print("No of actions taken -\n",
   counter)
63                  queue_child.queue.clear()
64                  break
65
66
67
```

```python
# Answer G(1)
#
# Taking any arbitrary initial state for goal state - 123804765
# For instance, let initial State - 213084675
# Cost - All moves - 1
# Answer -
# Path -
#  ['213084675', '213804675', '203814675', '023814675', '823014675', '823614075', '823614705', '823614750', '823610754',
#  '820613754', '802613754', '812603754', '812063754', '012863754', '102863754', '120863754', '123860754', '123864750',
#  '123864705', '123804765']
# No of actions taken - 31388
# Cost to Goal State -  19

import heapq

# defined variables

idx: int = 0
input_list = []
input_list_record = []
queue_child = []
goal_state = ['1', '2', '3', '8', '0', '4', '7', '6', '5']
counter = 0
child = []
cost = 0
parent = {}
path = []
child_string = ""
input_list_string = ""
cost_record = {}

# user input taken for puzzle piece sequence
while len(input_list) != 9:
    input_list = list(input("Please enter the puzzle piece numbers(9 digit long)"))
```

```
35      if len(input_list) != 9:
36          print("Entered number is not 9 digit")
37
38  # while len(goal_state) != 9:
39  #     goal_state = list(input("Please enter the Goal
    State(9 digit long)"))
40  #     if len(goal_state) != 9:
41  #         print("Entered number is not 9 digit")
42
43  u = 1  # int(input("Please enter UP Cost:"))
44  d = 1  # int(input("Please enter DOWN Cost:"))
45  r = 1  # int(input("Please enter RIGHT Cost:"))
46  l = 1  # int(input("Please enter LEFT Cost:"))
47
48  # index of directions list gives the possible actions
     on that index
49  directions = [[(1, r), (3, d)],
50                [(4, d), (0, l), (2, r)],
51                [(5, d), (1, l)],
52                [(0, u), (4, r), (6, d)], [(1, u), (5,
    r), (7, d), (3, l)],
53                [(2, u), (8, d), (4, l)],
54                [(3, l), (7, r)],
55                [(4, u), (8, r), (6, l)],
56                [(5, u), (7, l)]]
57
58  # used heapq function to create priority queue
59  # heap function used because of within queue value
    swappable functionality
60  heapq.heapify(queue_child)
61  heapq.heappush(queue_child, (0, input_list))
62
63  # initiated path to capture path to goal state as per
     UCS
64  parent[("".join(map(str, input_list)))] = None
65  cost_record[("".join(map(str, input_list)))] = 0
66
67  # loop to continue the game until goal state is
    achieved
68  while queue_child:
69      if input_list == goal_state:
```

```
70          print("Input State is already Goal State!")
71          break
72
73      (cost, input_list) = heapq.heappop(queue_child)
74      # Check if input node is goal state then print
    the required data
75      if input_list == goal_state:
76          input_list_record.append(input_list)
77          input_list_string = "".join(map(str,
    input_list))
78          counter += 1
79          # on achieving goal state, captured the path
     from goal state up to the initial state
80          s = input_list_string
81          while s is not None:
82              path.append(s)
83              s = parent[s]
84
85          print("Goal State -\n", child_string)
86          sequence_string = list(map(''.join,
    input_list_record))
87          print("Sequence -\n", sequence_string)
88          print("Path -\n", path[::-1])
89          print("No of actions taken -", counter)
90          print("Cost to Goal State - ", cost)
91          queue_child.clear()
92          break
93      # logic to generate UFS child node
94      # check to ensure input node not in closed list
95      if input_list not in input_list_record:
96          input_list_record.append(input_list)
97          idx = input_list.index('0')
98          counter += 1
99          input_list_string = "".join(map(str,
    input_list))
100         # generating child nodes from parent node
101         for i, j in directions[idx]:
102             child = input_list[:]
103             child[idx] = input_list[i]
104             child[i] = '0'
105             cost_i = cost + j
```

```
106             child_string = "".join(map(str, child))
107 
108             # pushing child node to queue
109             if child not in input_list_record:
110                 heapq.heappush(queue_child, (cost_i
    , child))
111                 heapq.heapify(queue_child)
112                 cost_record[child_string] = cost_i
113                 parent[child_string] =
    input_list_string
114             # check if the new child node generated
    is already in closed list but with larger cost
115 
116             else:
117                 if cost_i < cost_record[child_string
    ]:
118                     # replacing the already present
    child node with higher cost with the new same lower
    cost child node
119                     queue_child[queue_child.index((
    cost_record[child_string], child))] = (cost_i, child
    )
120                     parent[child_string] =
    input_list_string
121                     heapq.heapify(queue_child)
122 
```

```python
# Answer G(2)
#
# Taking any arbitrary initial state for goal state - 123804765
# For instance, let initial State - 213084675
# Cost - Up: 1, Down: 1, Left: 2, Right: 0.5
# Answer -
# Path -
#  ['213084675', '013284675', '103284675', '183204675', '183274605', '183274065', '183074265', '183704265', '103784265',
#  '013784265', '713084265', '713284065', '713284605', '713204685', '713024685', '013724685', '103724685', '123704685',
#  '123784605', '123784065', '123084765', '123804765']
# No of actions taken - 32047
# Cost to Goal State -  21.5

import heapq

# defined variables

idx: int = 0
input_list = []
input_list_record = []
queue_child = []
goal_state = ['1', '2', '3', '8', '0', '4', '7', '6', '5']
counter = 0
child = []
cost = 0
parent = {}
path = []
child_string = ""
input_list_string = ""
cost_record = {}

# user input taken for puzzle piece sequence
while len(input_list) != 9:
    input_list = list(input("Please enter the puzzle
```

```
34 piece numbers(9 digit long)"))
35     if len(input_list) != 9:
36         print("Entered number is not 9 digit")
37 
38 # while len(goal_state) != 9:
39 #     goal_state = list(input("Please enter the Goal
   State(9 digit long)"))
40 #     if len(goal_state) != 9:
41 #         print("Entered number is not 9 digit")
42 
43 u = 1  # int(input("Please enter UP Cost:"))
44 d = 1  # int(input("Please enter DOWN Cost:"))
45 r = 2  # int(input("Please enter RIGHT Cost:"))
46 l = 0.5  # int(input("Please enter LEFT Cost:"))
47 
48 # index of directions list gives the possible actions
    on that index
49 directions = [[(1, r), (3, d)],
50               [(4, d), (0, l), (2, r)],
51               [(5, d), (1, l)],
52               [(0, u), (4, r), (6, d)], [(1, u), (5,
   r), (7, d), (3, l)],
53               [(2, u), (8, d), (4, l)],
54               [(3, l), (7, r)],
55               [(4, u), (8, r), (6, l)],
56               [(5, u), (7, l)]]
57 
58 # used heapq function to create priority queue
59 # heap function used because of within queue value
   swappable functionality
60 heapq.heapify(queue_child)
61 heapq.heappush(queue_child, (0, input_list))
62 
63 # initiated path to capture path to goal state as per
    UCS
64 parent[("".join(map(str, input_list)))] = None
65 cost_record[("".join(map(str, input_list)))] = 0
66 
67 # loop to continue the game until goal state is
   achieved
68 while queue_child:
```

```
 69      if input_list == goal_state:
 70          print("Input State is already Goal State!")
 71          break
 72
 73      (cost, input_list) = heapq.heappop(queue_child)
 74      # Check if input node is goal state then print
    the required data
 75      if input_list == goal_state:
 76          input_list_record.append(input_list)
 77          input_list_string = "".join(map(str,
    input_list))
 78          counter += 1
 79          # on achieving goal state, captured the path
     from goal state up to the initial state
 80          s = input_list_string
 81          while s is not None:
 82              path.append(s)
 83              s = parent[s]
 84
 85          print("Goal State -\n", child_string)
 86          sequence_string = list(map(''.join,
    input_list_record))
 87          print("Sequence -\n", sequence_string)
 88          print("Path -\n", path[::-1])
 89          print("No of actions taken -", counter)
 90          print("Cost to Goal State - ", cost)
 91          queue_child.clear()
 92          break
 93      # logic to generate UFS child node
 94      # check to ensure input node not in closed list
 95      if input_list not in input_list_record:
 96          input_list_record.append(input_list)
 97          idx = input_list.index('0')
 98          counter += 1
 99          input_list_string = "".join(map(str,
    input_list))
100          # generating child nodes from parent node
101          for i, j in directions[idx]:
102              child = input_list[:]
103              child[idx] = input_list[i]
104              child[i] = '0'
```

```
105             cost_i = cost + j
106             child_string = "".join(map(str, child))
107 
108             # pushing child node to queue
109             if child not in input_list_record:
110                 heapq.heappush(queue_child, (cost_i
    , child))
111                 heapq.heapify(queue_child)
112                 cost_record[child_string] = cost_i
113                 parent[child_string] =
    input_list_string
114             # check if the new child node generated
    is already in closed list but with larger cost
115 
116             else:
117                 if cost_i < cost_record[child_string
    ]:
118                     # replacing the already present
    child node with higher cost with the new same lower
    cost child node
119                     queue_child[queue_child.index((
    cost_record[child_string], child))] = (cost_i, child
    )
120                     parent[child_string] =
    input_list_string
121                     heapq.heapify(queue_child)
122 
```