Assignment 3: Search (Given: 14 Feb 2023, Due: 7 Mar 2023)

General instructions

- Solutions are to be typed in the `.ipynb` file provided and uploaded in the lab course page in Moodle on the due date.
- Your code should be well commented and should be compatible with python3.
- For this assignment, you are allowed to import the libraries `random` and `queue` of python3. No other libraries may be imported.

In [1]:
```
import random as rnd
import queue
```

Generate Maze

A maze can be visualized as an arrangement of cells in a rectangular $m \times m$ grid with walls between some pairs of cells.

(a) Write a program that uses randomized depth-first search to generate a maze. The randomized depth-first search procedure is as follows: Starting from a given cell (say (0,0)), this algorithm produces a path that visits each cell in the grid according to the following recursive procedure.

- If the current cell C has a neighbouring cell that is not yet visited, chose one of such cells (say C') at random and remove the wall between these two cells. Repeat with C' as the current cell.
- If all neighbouring cells of the current cell C are already visited, then backtrack to the last cell $\widehat{C}$ with a neighbouring cell $\widehat{C}'$ that is not yet visited and repeat with $\widehat{C}'$ as the current cell.

A sample output of a 3×3 maze is along with the graph adjacency representation is as shown below. The bottom left corner of the maze is $(0,0)$ and top right corner of the maze is $(2,2)$.

In [2]:
```
class Cell:
  def __init__(self,x,y,m) -> None:
    # coordinate is same as index
    self.m = m
    # x coordinate
    self.x = x
    # y coordinate
    self.y = y
    # for printing path, we need parent
    self.parent = None
```

```
    # unique cell number for each cell
    self.cell_number = x*m + y
    # isVisited during creating maze
    self.isVisited = False
    # isVisited during searches
    self.isVisited_in_search = False
    # list of adjacent cells
    self.adjCells = []
    # during bfs, to store information of distance
    self.distance = 0
    # manhattan distance heuristic
    self.manhattan_distance_to_last_cell = (m-1-x) + (m-1-y)
    # x and y will always be less than or equal to m-1


    # if cell is neither in the corner nor the edge
    if x not in [0,m-1] and y not in [0,m-1]:
      adjCells = [
        (x - 1, y), # up ^
        (x,y+1), # right ->
        (x+1,y), # down v
        (x,y-1) # left
      ]
      self.adjCells = adjCells

    elif x not in [0,m-1] and y in [0,m-1]:
      # left and right edge

      if y == 0:
      # left edge
        adjCells = [
          (x - 1, y), # up ^
          (x,y+1), # right ->
          (x+1,y), # down v
          # () # left
        ]
        self.adjCells = adjCells
      else:
        # right edge
        adjCells = [
          (x - 1, y), # up ^
          # (), # right ->
          (x+1,y), # down v
          (x,y-1) # left
        ]
        self.adjCells = adjCells

    elif x  in [0,m-1] and y not in [0,m-1]:
      # top and bottom edge

      if x==0:
        # top edge
        adjCells = [
          # (), # up ^
          (x,y+1), # right ->
          (x+1,y), # down v
          (x,y-1) # left
```

```
            ]
            self.adjCells = adjCells
          else:
            adjCells = [
              (x - 1, y), # up ^
              (x,y+1), # right ->
              # (), # down v
              (x,y-1) # left
            ]
            self.adjCells = adjCells

      else:
        # corners
        if (x,y) == (0,0):
          # top left corner
          adjCells = [
            # (), # up ^
            (x,y+1), # right ->
            (x+1,y), # down v
            # () # left
          ]
          self.adjCells = adjCells
        elif (x,y) == (0,m-1):
          # top right corner
          adjCells = [
            # (), # up ^
            # (), # right ->
            (x+1,y), # down v
            (x,y-1) # left
          ]
          self.adjCells = adjCells
        elif (x,y) == (m-1,m-1):
          # bottom right corner
          adjCells = [
            (x - 1, y), # up ^
            # (), # right ->
            # (), # down v
            (x,y-1) # left
          ]
          self.adjCells = adjCells
        else:
          # bottom left corner
          adjCells = [
            (x - 1, y), # up ^
            (x,y+1), # right ->
            # (), # down v
            # () # left
          ]
          self.adjCells = adjCells

      adjCell_numbers = [
        adjCell[0]*m + adjCell[1] for adjCell in self.adjCells
      ]
      self.adjCell_numbers = list(adjCell_numbers)
      # the unique cell numbers of adj Cells

      # list of walled neighbours
```

```
      self.walled_neighbours = list(self.adjCell_numbers)
      # list of non walled neighbours
      self.non_walled_neighbours = [
        x for x in self.adjCell_numbers if x not in self.walled_neighbours
      ]

  def generate_children_of_cell(self):
    # returns all cells which are adjacent to the current cell, whether w
    return self.adjCell_numbers

  def generate_non_walled_neighbours(self):
    # returns list of non-walled neighbours, the list contains the unique
    return self.non_walled_neighbours

  def manhattan_distance_heuristic(self) -> int:
    # returns manhattan distance heuristic
    return self.manhattan_distance_to_last_cell

  def print_current_cell(self):
    # utility function to print the attributes of the current cell
    print()
    print(f'the cell is {(self.x,self.y)}')
    print(f'the cell number is {self.cell_number}')
    print(f'the adj Cells are {self.adjCells}')
    print(f'the adjCell numbers are {self.adjCell_numbers}')
    print(f'the walled neighbours are {self.walled_neighbours}')
    print(f'the non-walled neighbours are {self.non_walled_neighbours}')
    print()

  def print_adjacency_list(self):
    # prints the list of non-walled neighbours of the current cell
    print()
    x = self.non_walled_neighbours
    y = [
      (int(cell_number/self.m),cell_number%self.m)
      for cell_number in x
    ]
    print(f'Node {(self.x,self.y)}: ', end=' ')
    for coord in y:
      print(coord,end=' ')
    print()
    print()

  # the below function is for the purpose of priority queue, when using p
  def __lt__(self,other):
    return self.manhattan_distance_to_last_cell < other.manhattan_distanc
```

In [3]:

```
class Maze:
  def __init__(self,m) -> None:
    # maze side length
    self.sides = m
    # list of 'Cell' Objects
    self.cells = [Cell(i,j,m) for i in range(m) for j in range(m)]
    # creating an empty dictionary to store all the cells with key as the
    # the reason for using dictionary is to access the elements fast
    self.cell_dictionary = dict()
```

```
    for cell in self.cells:
      # the key of each cell is of the form 'cell_{cell number}'
      self.cell_dictionary[f'cell_{cell.cell_number}'] = cell
    # self.created_maze = self.generate_maze()
    # we call generate_maze() function to generate a random maze
    self.generate_maze()
    # maze_img is a list of list which is used to print the grid of maze
    self.maze_img = [
      list(['' for _ in range(2*m + 1)]) for _ in range(2*m + 1)
    ]
    # calling create_maze_img() function to create the maze image
    self.create_maze_img()
    # since list is mutable, we have to create new list in every row

  def generate_maze(self):
    # we are generating maze using randomized depth first search
    self.dfs_create_maze(self.cell_dictionary['cell_0'])

  def child_generator(self,cell: Cell):
    # returns the adjacent cells of the cell, the list will contain the c
    return cell.generate_children_of_cell()

  def dfs_create_maze(self,cell: Cell):
    # create maze function

    # length of maze
    m = self.sides

    # adj cells of the current cell
    adj_cell_number_list = list(cell.generate_children_of_cell())
    # adj_cells = cell.generate_children_of_cell().copy()

    # we are shuffling it in order to simulate randomized picking of cell
    rnd.shuffle(adj_cell_number_list)

    # isVisited in creating maze is set to True
    cell.isVisited = True

    # for each cell in the adj cells
    for cell_to_be_considered in adj_cell_number_list:
      cell_to_be_considered: int

      # using a try except block in order to catch any possible errors
      try:
        cellCheck = self.cell_dictionary[f'cell_{cell_to_be_considered}']
      except:
        print(f'the key {cell_to_be_considered} not found')
      cellCheck: Cell
      if cellCheck.isVisited == False:
        # if cell is not visited
        # first remove wall between this cell and the picked cell
        try:
          cell.walled_neighbours.remove(cellCheck.cell_number)
        except:
          print(f'cell {cellCheck.cell_number} not found in {cell.cell_nu
        cell.non_walled_neighbours.append(cellCheck.cell_number)
        # now remove wall between picked cell and this cell
```

```
        try:
          cellCheck.walled_neighbours.remove(cell.cell_number)
        except:
          print(f'cell {cell.cell_number} not found in {cellCheck.cell_nu
        cellCheck.non_walled_neighbours.append(cell.cell_number)

        # then call dfs_create_maze with the non visited cell
        self.dfs_create_maze(cellCheck)
    del adj_cell_number_list
    # in order to save memory





  def print_cells_of_maze(self):
    # utility function to print the cell number of cells of maze in the d
    for key,value in self.cell_dictionary.items():
      value: Cell
      print(key,value.cell_number)


  def create_maze_img(self):
    # in order to create the maze image
    m = self.sides
    for i in range(2*self.sides + 1):
      if i%2 == 0:
        for j in range(2*self.sides + 1):

          if i==0 or i==2*self.sides:
            # borders
            if j%2 == 0:
              self.maze_img[i][j] = '+'
            else:
              self.maze_img[i][j] = '---'
          # now we have to check if the neighbours are walled
          else:


            if j%2 == 0:
              self.maze_img[i][j] = '+'
            else:
              # we can check the list of non walled neighbours


              # this will also work if the cell is near the wall because
              (x,y) = (int((i-1)/2),int((j-1)/2))
              cNum = x*m + y
              # to check if the below cell is walled in order to decide w
              is_below_cell_walled = True
              try:
                current_cell = self.cell_dictionary[f'cell_{cNum}']
              except:
                print(f'the key {cNum} not found ')
              current_cell: Cell
              if cNum+m in current_cell.non_walled_neighbours:
                # if below cell is present in non walled neighbours, we u
```

```
                is_below_cell_walled = False


              if is_below_cell_walled:
                self.maze_img[i][j] = '---'
              else:
                self.maze_img[i][j] = '   '


      else:
        for j in range(2*self.sides + 1):
          (x,y) = (int((i-1)/2),int((j-1)/2))
          if j==0 or j==2*self.sides:
            # for left and right boundaries
            self.maze_img[i][j] = '|'
          # if j%2 == 0:
          #   self.maze_img[i][j] = '|'
          else:

            if j%2 == 1:
              self.maze_img[i][j] = ''
            else:
              (x,y) = (int((i-1)/2),int((j-1)/2))
              cNum = x*m + y
              # to check if the right cell is walled in order to decide w
              is_right_cell_walled = True
              # current_cell = self.cell_dictionary[f'cell_{cNum}']
              try:
                current_cell = self.cell_dictionary[f'cell_{cNum}']
              except:
                print(f'the key {cNum} not found ')
              current_cell: Cell
              if cNum+1 in current_cell.non_walled_neighbours:
                # if right cell is in non-walled neighbours, we update is
                is_right_cell_walled = False

              # self.maze_img[i][j] = '|'

              if is_right_cell_walled:
                self.maze_img[i][j] = '|'
              else:
                self.maze_img[i][j] = ''




  def print_maze(self):
    # utility function to print the adjacency list of maze cells and imag
    for key, cell in self.cell_dictionary.items():
      cell: Cell
      # cell.print_current_cell()
      cell.print_adjacency_list()
    for i in range(2*self.sides + 1):
      if i%2 == 0:
        for j in range(2 * self.sides + 1):
          if j%2 == 0:
            # print('+',end='')
```

```
            print(self.maze_img[i][j],end='')
          else:
            # print('---',end='')
            print(self.maze_img[i][j],end='')
        print()
      else:
        for j in range(2 * self.sides + 1):
          # if j==0 or j==2*self.sides:
          if j%2 == 0:
            # print('|',end=' ')
            print(self.maze_img[i][j],end=' '  if self.maze_img[i][j] ==
          else:
            # print('',end='  ')
            print(self.maze_img[i][j],end='  ')
        print()
```

In [4]:

```
class Agent:
  def __init__(self,maze: Maze) -> None:
    # which maze is the agent present in
    self.in_maze = maze
    # initial location of the agent is [0,0]
    self.initial_loc = [0,0]
    # attribute to count the number of dfs_explore
    self.dfs_explore_count = 0
    # attribute to count the number of bfs_explore
    self.bfs_explore_count = 0
    # attribute to count the number of astar_explore
    self.astar_explore_count = 0

  def is_goal(self,cellNum):
    # if the agent is in the last cell, then goal state has reached
    m = self.in_maze.sides
    if cellNum == m*m - 1:
      return True
    return False


  def dfs_search(self,print_path = True):
    # initializing dfs_explore count to zero
    self.dfs_explore_count = 0
    maze = self.in_maze
    cell_dictionary = maze.cell_dictionary
    # initializing the parent of each cell to None and is Visited in sear
    for i in range(maze.sides * maze.sides):
      cell = cell_dictionary[f'cell_{i}']
      cell: Cell
      cell.parent = None
      cell.isVisited_in_search = False
    # for _,cell in cell_dictionary.items():
    for i in range(maze.sides * maze.sides):
      cell = cell_dictionary[f'cell_{i}']
      cell: Cell
      isPathFound = False
      # isPathFound is to check if path has been found
      # if path is found, then we can stop the dfs there
      if cell.isVisited_in_search == False:
```

```
        # print(f'Going to cell {cell.cell_number}')
        isPathFound = self.dfs_explore(cell)

        # self.dfs_explore(cell)
      if isPathFound == True:
        if print_path:
          print('Path found from initial cell to final cell using DFS')
          # self.print_path(cell_dictionary[f'cell_{maze.sides * maze.sid
          self.print_path(cell_dictionary[f'cell_{maze.sides * maze.sides
          print()
          print(f'the number of cells explored are {self.dfs_explore_coun
        # break
        return
    # self.print_path(cell_dictionary[f'cell_{maze.sides * maze.sides - 1

  def dfs_explore(self,current_cell: Cell):
    # print(f'coming to cell {current_cell.cell_number}')

    # dfs explore count increases whenever it comes to this function call
    self.dfs_explore_count += 1

    # updating is visited in search to True
    current_cell.isVisited_in_search = True


    maze = self.in_maze
    cell_dictionary = maze.cell_dictionary

    # taking the non-walled neighbours because agent can only go to those
    adj_non_walled_neighbours = current_cell.non_walled_neighbours

    # initializing isPath_found to False
    isPath_found = False

    for adjCellNum in adj_non_walled_neighbours:
      try:
        adjCell = cell_dictionary[f'cell_{adjCellNum}']
      except:
        print(f'the key {adjCellNum} is not found')
      adjCell: Cell
      if adjCell.isVisited_in_search == False:
        # if adj non-walled neighbour is unvisited we set the parent of t
        adjCell.parent = current_cell

        if self.is_goal(adjCellNum):
          # if that cell is a goal state

          # print(f'Going to cell {adjCell.cell_number}')
          # self.dfs_explore_count += 1
          # print('Path found')
          # print('Path found from initial cell to final cell using DFS')
          # self.print_path(adjCell)
          # print()

          # if that cell is a goal state, we return True
          return True
        else:
```

```
            # print(f'Going to cell {adjCell.cell_number}')

            # we set the isPath to the value coming from the next function
            isPath_found = self.dfs_explore(adjCell)
            if isPath_found:
              return isPath_found

            # print(f'Returned to {current_cell.cell_number}')
            # return isPath_found
            continue
      # print(f'Returning from cell {current_cell.cell_number}')

      # we have to return isPath found from each dfs_explore calls
      return isPath_found

    def bfs_search(self,print_path = True):
      # print('in bfs search')
      # initializing explore count to zero
      self.bfs_explore_count = 0
      maze = self.in_maze
      cell_dictionary = maze.cell_dictionary

      # initializing each cell's parent to None and distance to -1 and isVi
      for i in range(maze.sides * maze.sides):
        cell = cell_dictionary[f'cell_{i}']
        cell: Cell
        cell.parent = None
        cell.isVisited_in_search = False
        cell.distance = -1

      # using a queue for dfs
      q = queue.Queue()

      # source cell is the cell from where we start bfs, in this case initi
      source_cell = cell_dictionary[f'cell_{0}']
      source_cell: Cell
      # q.put(source_cell,timeout=0.0045)
      q.put(source_cell)
      # q.put_nowait(source_cell)
      source_cell.distance = 0
      # Use qsize() == 0 as a direct substitute, but be aware that either a
      # the above was a warning which came when trying to use while q.empty
      while not q.qsize() == 0:
        # print('executing while loop')
        # u_cell = q.get(timeout=0.0045)
        # u_cell = q.get_nowait()

        # to verify its not a priority queue

        # print('printing queue')
        # for x in q.queue:
        #   x: Cell
        #   print(x.manhattan_distance_to_last_cell,end=' ')
        # print()

        # dequeueing the first cell in the queue
        u_cell = q.get()
```

```
      # a cell gets explored only when it comes to while loop
      self.bfs_explore_count += 1
      u_cell: Cell

      # updating the isvisited in search to True
      u_cell.isVisited_in_search = True

      # checking the non-walled neighbours of the current cell
      adj_cell_numbers = u_cell.generate_non_walled_neighbours()

      # for each non walled cell
      for cellNum in adj_cell_numbers:
        # cell_to_check = cell_dictionary[f'cell_{cellNum}']
        try:
          cell_to_check = cell_dictionary[f'cell_{cellNum}']
        except:
          print(f'the key {cellNum} not found')
        cell_to_check: Cell
        if cell_to_check.distance == -1:
          # if distance is -1, it means, it has not been visited yet
          cell_to_check.distance = u_cell.distance + 1
          q.put(cell_to_check)
          # enqueuing into the queue

          # q.put(cell_to_check,timeout=0.0045)
          # q.put_nowait(cell_to_check)
          cell_to_check.parent = u_cell

          # checking if the cell is a goal state
          if self.is_goal(cellNum):
            if print_path:
              print('Path Found from initial cell to final cell using BFS
              self.print_path(cell_to_check)
              print()
              print(f'the number of vertices explore is {self.bfs_explore
            del q
            return
      # return
    # del q
    # return


  def astar_search(self,print_path = True):
    # initializing explore count to zero
    self.astar_explore_count = 0
    maze = self.in_maze
    cell_dictionary = maze.cell_dictionary

    # initializing each cell's parent to None and distance to -1 and isVi
    for i in range(maze.sides * maze.sides):
      cell = cell_dictionary[f'cell_{i}']
      cell: Cell
      cell.parent = None
      cell.isVisited_in_search = False
      cell.distance = -1

    # using a priority queue
```

```
    q = queue.PriorityQueue()
    source_cell = cell_dictionary[f'cell_{0}']
    source_cell: Cell
    source_cell.distance = 0
    # g function is initially zero
    g_function = 0

    # entries into the queue are of the form (priority number,data)
    # priority queue is based on f = g + h
    q.put((source_cell.manhattan_distance_heuristic()+g_function,source_c
    while not q.qsize() == 0:

      # to verify its a priority queue
      # print('printing queue')
      # for x in q.queue:
      #   print(x[0],end=' ')
      # print()

      u_cell = q.get()[1]
      # print(u_cell)
      self.astar_explore_count += 1
      # only when we come to this while loop will we increment the explor
      u_cell: Cell
      u_cell.isVisited_in_search = True
      # taking the adj cells which are non walled
      adj_cell_numbers = u_cell.generate_non_walled_neighbours()

      # for each cell in adj non walled neighbours
      for cellNum in adj_cell_numbers:
        try:
          cell_to_check = cell_dictionary[f'cell_{cellNum}']
        except:
          print(f'the key {cellNum} not found')
        cell_to_check: Cell
        if cell_to_check.distance == -1:
          cell_to_check.distance = u_cell.distance + 1
          # g function is the distance
          to_add = ((cell_to_check.manhattan_distance_heuristic() + cell_

          q.put(to_add)
          # updating the parent cell
          cell_to_check.parent = u_cell
          if self.is_goal(cellNum):
            # checking if its a goal state
            if print_path:
              print('Path Found from initial cell to final cell using A*'
              self.print_path(cell_to_check)
              print()
              print(f'the number of vertices explored is {self.astar_expl
            del q
            return


  def print_path(self,cell: Cell):
    # this is the recursive print path function
    if not cell:
      # if cell is None, then return
```

```
        return
      # else print path of parent and then print the current cell
      self.print_path(cell.parent)
      print((cell.x,cell.y),end=' ')

    def print_parent(self):
      # utility function to print the parent of each cell
      cell_dictionary = self.in_maze.cell_dictionary
      for _,cell in cell_dictionary.items():
        cell: Cell
        parent = cell.parent
        parent: Cell

        print(cell.cell_number, 'parent is', parent.cell_number if parent e
```

In [5]:

```
# creating a new maze of side length 3
maze = Maze(3)
# maze.print_cells_of_maze()
maze.print_maze()
# printing the adjacency list of maze cells and the image of maze

search_agent = Agent(maze)
```

```
Node (0, 0):  (1, 0)


Node (0, 1):  (0, 2)


Node (0, 2):  (1, 2) (0, 1)


Node (1, 0):  (0, 0) (1, 1)


Node (1, 1):  (1, 0) (1, 2)


Node (1, 2):  (1, 1) (0, 2) (2, 2)


Node (2, 0):  (2, 1)


Node (2, 1):  (2, 2) (2, 0)


Node (2, 2):  (1, 2) (2, 1)

+---+---+---+
|   |       |
+   +---+   +
|           |
+---+---+   +
|           |
+---+---+---+
```

(b) Write a program to do DFS on a $m \times m$ maze given in adjacency representation to find a route from the source cell $(0, 0)$ to the destination cell $(m-1, m-1)$. Output the route and the number of cells explored.

In [6]:
```
search_agent.dfs_search()
```

```
Path found from initial cell to final cell using DFS
(0, 0) (1, 0) (1, 1) (1, 2) (2, 2)
the number of cells explored are 6
```

(c) Write a program to do BFS on a $m \times m$ maze given in adjacency representation to find a route from the source cell $(0, 0)$ to the destination cell $(m-1, m-1)$. Output the route and the number of cells explored.

In [7]:
```
search_agent.bfs_search()
```

```
Path Found from initial cell to final cell using BFS
(0, 0) (1, 0) (1, 1) (1, 2) (2, 2)
the number of vertices explore is 4
```

(d) Write a program to do A* search on a $m \times m$ maze given in adjacency representation to find a route from the source cell $(0, 0)$ to the destination cell $(m-1, m-1)$. Use the Manhattan heuristic for A* search. The Manhattan distance between two cells of the maze (i, j) and (k, ℓ) where $i, j, k, \ell \in \{0, 1 \ldots, m-1\}$ is $|i-k| + |j-\ell|$. Output the route and the number of cells explored.

In [8]:
```
search_agent.astar_search()
```

```
Path Found from initial cell to final cell using A*
(0, 0) (1, 0) (1, 1) (1, 2) (2, 2)
the number of vertices explored is 4
```

In [9]:
```
# to see the efficiency of dfs,bfs and astar

bfs_explore = 0
dfs_explore = 0
astar_explore = 0


for i in range(1000):
  # maze_new = Maze(30)
  maze_new = Maze(10)
  agent_new = Agent(maze_new)
  agent_new.dfs_search(print_path=False)
  dfs_explore += agent_new.dfs_explore_count
  agent_new.bfs_search(print_path=False)
  bfs_explore += agent_new.bfs_explore_count
  agent_new.astar_search(print_path=False)
  astar_explore += agent_new.astar_explore_count
  del agent_new
  del maze_new

print(f'average number of vertices DFS explored: {dfs_explore/1000}')
print(f'average number of vertices BFS explored: {bfs_explore/1000}')
print(f'average number of vertices A * explored: {astar_explore/1000}')
```

```
average number of vertices DFS explored: 64.866
average number of vertices BFS explored: 60.707
average number of vertices A * explored: 54.101
```

Sliding Blocks

Consider the sliding blocks puzzle where we are given a 3×3 grid of blocks with each block containing a unique integer between 0 and 8. An example configuration is given below.

1 2 3

7 8 5

0 6 4

At each step, the block containing 0 can swap places with an adjacent block.

In [10]:
```
# it is required that we generate a maze which is solvable

def random_index(m: int):
  x = rnd.randrange(0,m)
  y = rnd.randrange(0,m)
  return (x,y)

def absolute_value(x: int):
  return x if x>=0 else -x
```

In [11]:
```
# for different heuristics we have to change the __lt__ function of the c

class Grid_State:
  def __init__(self,grid: list) -> None:

    self.grid = grid
    self.parent = None
    manhattan_distance_heuristic = 0
    m = len(grid)
    self.length_of_grid = m
    num = 1
    self.depth_distance = -1
    # goal_grid = [
    #   list([0 for _ in range(m)]) for _ in range(m)
    # ]
    # for i in range(m):
    #   for j in range(m):
    #     if (i,j) != (m-1,m-1):
    #       goal_grid[i][j] = num
    #       num += 1
    number_to_index_dictionary = dict()
    num = 1
    for i in range(m):
      for j in range(m):
        if num!=m*m:
          number_to_index_dictionary[num] = (i,j)
          num += 1
    number_to_index_dictionary[0] = (m-1,m-1)


    for i in range(m):
      for j in range(m):
        number_in_i_j = grid[i][j]
        actual_pos_of_number = number_to_index_dictionary[number_in_i_j]
        manhattan_distance_heuristic += absolute_value(i-actual_pos_of_nu



    self.manhattan_distance_heuristic = manhattan_distance_heuristic
    del number_to_index_dictionary
    self.children_state_of_this_state = list()
    self.number_of_misplaced_blocks = self.compute_number_of_mispaced_til
    self.manhattan_distance_heuristic_of_zero = self.compute_manhattan_di

  # def compute_manhattan_distance_heuristic(self):

  def __lt__(self,other):
    # return True
    return self.manhattan_distance_heuristic < other.manhattan_distance_h
    # return self.number_of_misplaced_blocks < other.number_of_misplaced_
    # return self.manhattan_distance_heuristic_of_zero < other.manhattan_

  def compute_number_of_mispaced_tiles(self):
    grid = self.grid
    count = 0
    m = self.length_of_grid
    number = 1
```

```
    for i in range(m):
      for j in range(m):
        if (i,j) == (m-1,m-1):
          if grid[i][j] != 0:
            count += 1
        else:
          if grid[i][j] != number:
            count += 1
          number += 1
    return count

  def compute_manhattan_distance_of_zero(self):
    grid = self.grid
    x = 0
    y = 0
    m = self.length_of_grid
    for i in range(m):
      for j in range(m):
        if grid[i][j] == 0:
          x = i
          y = i
          # since m-1 will always be greater than or equal to x and y
          return (m-1)-x + (m-1) - y
```

In [12]:

```
class Sliding_Block_Grid:
  def __init__(self, m=3,input_puzzle = True,depth_for_generating_sovable
    self.sides =m
    self.rows = m
    self.columns = m
    self.pos_zero = [0,0]
    self.grid = [
      list([-1 for _ in range(m)]) for _ in range(m)
    ]
    self.depth = depth_for_generating_sovable_maze

    # to create new list in each row

    # number = 0
    # while number < m*m:
    #   i,j = random_index(m)
    #   if self.grid[i][j] == -1:
    #     self.grid[i][j] = number
    #     if number == 0:
    #       self.pos_zero = [i,j]
    #     number += 1


    if input_puzzle == True:
      self.grid = [
        list([int(input()) for _ in range(m)]) for _ in range(m)
      ]
    elif input_puzzle == False:
      self.create_solvable_puzzle()

    self.current_grid_state = Grid_State(self.grid)
    # list of states
```

```
    # self.solution = list()


  def create_solvable_puzzle(self):
    # to create solvable puzzle
    m = self.sides
    grid = [
      list([0 for _ in range(m)]) for _ in range(m)
    ]
    number = 1
    for i in range(m):
      for j in range(m):
        if (i,j) == (m-1,m-1):
          grid[i][j] = 0
        else:
          grid[i][j] = number
          number+=1

    # depth = rnd.randrange(10,20)
    # depth = int(input())
    depth = self.depth
    i = m-1
    j = m-1
    # continue from here
    for _ in range(depth):
      actionList = self.possible_actions_to_create_solvable_maze(i,j,m)
      action = rnd.choices(actionList,k=1)[0]
      if action=='u':
        grid[i][j], grid[i-1][j] = grid[i-1][j], grid[i][j]
        i,j = i-1,j
      elif action=='d':
        grid[i][j], grid[i+1][j] = grid[i+1][j], grid[i][j]
        i,j = i+1,j
      elif action=='l':
        grid[i][j], grid[i][j-1] = grid[i][j-1], grid[i][j]
        i,j = i,j-1
      elif action == 'r':
        grid[i][j], grid[i][j+1] = grid[i][j+1], grid[i][j]
        i,j = i,j+1
      else:
        pass
    self.grid = grid



  def possible_actions_to_create_solvable_maze(self,i: int, j:int, m:int)
    # neither corner nor edge
    if i not in [0,m-1] and j not in [0,m-1]:
      return ['u','r','d','l']
    # edge
    elif i not in [0,m-1] and j in [0,m-1]:
      if j==0:
        # left edge
        return ['u','r','d']
      else:
        # right edge
        return ['u','d','l']
```

```
    elif i in [0,m-1] and j not in [0,m-1]:
      if i==0:
        # top edge
        return ['r','d','l']
      else:
        # bottom edge
        return ['u','r','l']
    # corners
    else:
      if (i,j)==(0,0):
        # top left
        return ['r','d']
      elif (i,j)==(0,m-1):
        # top right
        return ['d','l']
      elif (i,j)==(m-1,m-1):
        # bottom right
        return ['u','l']
      else:
        # bottom left
        return ['u','r']




  def print_current_state_of_grid(self):
    print()
    for i in range(self.sides):
      for j in range(self.sides):
        print(self.grid[i][j],end=' ')
      print()
    # print(f'position of zero is {self.pos_zero}')
    print()

  def is_goal_state(self,grid: list):
    number = 1
    m = self.sides
    for i in range(m):
      for j in range(m):
        if (i,j) != (m-1,m-1) and grid[i][j] == number:
          number += 1
        elif (i,j) == (m-1,m-1):
          if grid[i][j] != 0:
            return False
        else:
          return False
    return True
```

In [13]:
```
class Agent:
  def __init__(self,puzzle_grid: Sliding_Block_Grid) -> None:

    self.in_puzzle = puzzle_grid
    self.in_grid = puzzle_grid.grid.copy()
    self.length_of_puzzle = puzzle_grid.sides
    self.pos_agent = puzzle_grid.pos_zero
    # need not write list(puzzle_grid.pos_zero) because whenever pos_zero
```

```
    # since list is call by reference
    self.grid_state = Grid_State(puzzle_grid.grid.copy())
    # grid state is initialized with a copy of the list
    # so any modification in this list will not affect in the original li
    self.solution_list = list()
    self.astar_explore_count_manhattan_distance = 0
    self.astar_explore_count_number_of_misplaced_blocks = 0
    self.astar_explore_count_manhattan_distance_of_zero = 0


  def possible_actions(self):
    m = self.length_of_puzzle
    i = self.pos_agent[0]
    j = self.pos_agent[1]
    # neither corner nor edge
    if i not in [0,m-1] and j not in [0,m-1]:
      return ['u','r','d','l']
    # edge but not corner
    elif i in [0,m-1] and j not in [0,m-1]:
      if i==0:
        # top edge
        return ['r','d','l']
      else:
        # bottom edge
        return ['r','l','u']
    elif i not in [0,m-1] and j in [0,m-1]:
      if j==0:
        # left edge
        return ['u','r','d']
      else:
        # right edge
        return ['u','d','l']
    # corner
    else:
      if (i,j) == (0,0):
        # top left
        return ['r','d']
      if (i,j) == (0,m-1):
        # top right
        return ['d','l']
      if (i,j) == (m-1,m-1):
        # bottom right
        return ['u','l']
      else:
        # bottom left
        return ['u','r']

  def possible_actions_from_state(self,grid_state: Grid_State):
    m = self.length_of_puzzle
    pos_agent = self.find_pos_zero(grid_state)

    i = pos_agent[0]
    j = pos_agent[1]
    # neither corner nor edge
    if i not in [0,m-1] and j not in [0,m-1]:
      return ['u','r','d','l']
    # edge but not corner
```

```
    elif i in [0,m-1] and j not in [0,m-1]:
      if i==0:
        # top edge
        return ['r','d','l']
      else:
        # bottom edge
        return ['r','l','u']
    elif i not in [0,m-1] and j in [0,m-1]:
      if j==0:
        # left edge
        return ['u','r','d']
      else:
        # right edge
        return ['u','d','l']
    # corner
    else:
      if (i,j) == (0,0):
        # top left
        return ['r','d']
      if (i,j) == (0,m-1):
        # top right
        return ['d','l']
      if (i,j) == (m-1,m-1):
        # bottom right
        return ['u','l']
      else:
        # bottom left
        return ['u','r']

  def is_goal_state(self,grid_state: Grid_State):
    number = 1
    m = self.length_of_puzzle
    grid = grid_state.grid
    for i in range(m):
      for j in range(m):
        if (i,j) != (m-1,m-1) and grid[i][j] == number:
          number += 1
        elif (i,j) == (m-1,m-1):
          if grid[i][j] != 0:
            return False
        else:
          return False
    return True

  def create_copy(self,puzzle_grid: list):
    copy = [None for _ in range(self.length_of_puzzle)]
    for i in range(self.length_of_puzzle):
      copy[i] = list(puzzle_grid[i])
    return copy



  def find_pos_zero(self,grid_state: Grid_State):
    grid = grid_state.grid
    for i in range(self.length_of_puzzle):
      for j in range(self.length_of_puzzle):
        if grid[i][j] == 0:
```

```
        return (i,j)

  def child_generator(self) -> Grid_State:
    current_state_of_grid = self.in_puzzle.grid
    copy_of_current_state_of_grid = self.create_copy(current_state_of_gri
    possible_actions = self.possible_actions()
    num_possible_actions = len(possible_actions)
    children_list = [
      self.create_copy(copy_of_current_state_of_grid) for _ in range(num_
    ]
    for i in range(num_possible_actions):
      self.do_action(children_list[i],possible_actions[i])
    children_states = [
      Grid_State(child_grid) for child_grid in children_list
    ]
    return children_states

  def child_generator_from_state(self, grid_state: Grid_State):
    grid = list(grid_state.grid)
    possible_actions = self.possible_actions_from_state(grid_state)
    num_possible_actions = len(possible_actions)
    children_list = [
      self.create_copy(grid) for _ in range(num_possible_actions)
    ]
    children_states = [
      Grid_State(child_list) for child_list in children_list
    ]
    for i in range(num_possible_actions):
      self.do_action_from_state(children_states[i],possible_actions[i])
    return children_states

  def do_action_from_state(self,grid_state: Grid_State, action: str):
    pos_agent = self.find_pos_zero(grid_state)
    i = pos_agent[0]
    j = pos_agent[1]
    grid = grid_state.grid
    if action == 'u':
      grid[i][j], grid[i-1][j] = grid[i-1][j], grid[i][j]
    elif action == 'r':
      grid[i][j], grid[i][j+1] = grid[i][j+1], grid[i][j]
    elif action == 'd':
      grid[i][j], grid[i+1][j] = grid[i+1][j], grid[i][j]
    elif action == 'l':
      grid[i][j], grid[i][j-1] = grid[i][j-1], grid[i][j]
    else:
      pass

  def do_action(self,grid: list, action: str):
    i = self.pos_agent[0]
    j = self.pos_agent[1]
    if action == 'u':
      grid[i][j], grid[i-1][j] = grid[i-1][j], grid[i][j]
    elif action == 'r':
      grid[i][j], grid[i][j+1] = grid[i][j+1], grid[i][j]
    elif action == 'd':
      grid[i][j], grid[i+1][j] = grid[i+1][j], grid[i][j]
    elif action == 'l':
```

```
      grid[i][j], grid[i][j-1] = grid[i][j-1], grid[i][j]
    else:
      pass

  def astar_search_using_manhattan(self,print_route = True):
    self.astar_explore_count_manhattan_distance = 0
    initial_state = self.grid_state
    # create a queue
    q = queue.PriorityQueue()
    # tuple is priority number, data
    g_function = 0
    # depth distance is the g function
    initial_state.depth_distance = 0
    # priority is based on f function = g+h
    tuple_to_be_added = (initial_state.manhattan_distance_heuristic + g_f
    q.put(tuple_to_be_added)

    if self.is_goal_state(initial_state) or initial_state.manhattan_dista
      if print_route:
        print('Solution found')
        self.print_solution(initial_state)
      return


    while not q.qsize == 0:
      self.astar_explore_count_manhattan_distance += 1
      u_state = q.get()[1]

      u_state: Grid_State
      children_states = self.child_generator_from_state(u_state)
      for child_state in children_states:
        child_state: Grid_State
        child_state.parent = u_state
        if self.is_goal_state(child_state) or child_state.manhattan_dista
          if print_route:
            print('Solution found')
            self.print_solution(child_state)
          return
        child_state.depth_distance = u_state.depth_distance + 1
        tuple_to_be_added_to_priority_queue = (child_state.manhattan_dist
        # print('not yet found')
        q.put(tuple_to_be_added_to_priority_queue)

  def astar_search_using_number_of_misplaced_blocks(self,print_route = Tr
    self.astar_explore_count_number_of_misplaced_blocks = 0
    initial_state = self.grid_state
    initial_state.depth_distance = 0
    # create a queue
    q = queue.PriorityQueue()
    # tuple is priority number, data
    tuple_to_be_added = (initial_state.number_of_misplaced_blocks + initi
    q.put(tuple_to_be_added)

    if self.is_goal_state(initial_state) or initial_state.number_of_mispl
      if print_route:
        print('Solution found')
        self.print_solution(initial_state)
```

```
      return


    while not q.qsize == 0:
      self.astar_explore_count_number_of_misplaced_blocks += 1
      u_state = q.get()[1]

      u_state: Grid_State
      children_states = self.child_generator_from_state(u_state)
      for child_state in children_states:
        child_state: Grid_State
        child_state.parent = u_state
        if self.is_goal_state(child_state) or child_state.number_of_mispl
          if print_route:
            print('Solution found')
            self.print_solution(child_state)
          return
        child_state.depth_distance = u_state.depth_distance + 1
        tuple_to_be_added_to_priority_queue = (child_state.number_of_misp
        # print('not yet found')
        q.put(tuple_to_be_added_to_priority_queue)



  def astar_search_using_manhattan_distance_of_zero(self,print_route = Tr
    self.astar_explore_count_manhattan_distance_of_zero = 0
    initial_state = self.grid_state
    # create a queue
    q = queue.PriorityQueue()
    # tuple is priority number, data
    initial_state.depth_distance = 0
    tuple_to_be_added = (initial_state.manhattan_distance_heuristic_of_ze
    q.put(tuple_to_be_added)


    if self.is_goal_state(initial_state):
      if print_route:
        print('Solution found')
        self.print_solution(initial_state)
      return


    while not q.qsize == 0:
      self.astar_explore_count_manhattan_distance_of_zero += 1
      u_state = q.get()[1]

      u_state: Grid_State
      children_states = self.child_generator_from_state(u_state)
      for child_state in children_states:
        child_state: Grid_State
        child_state.parent = u_state
        if self.is_goal_state(child_state):
          if print_route:
            print('Solution found')
            self.print_solution(child_state)
          return
        child_state.depth_distance = u_state.depth_distance + 1
```

```
            tuple_to_be_added_to_priority_queue = (child_state.manhattan_dist
            # print('not yet found')
            q.put(tuple_to_be_added_to_priority_queue)




    def print_grid(self,grid_state: Grid_State):
      print()
      grid = grid_state.grid
      for i in range(self.length_of_puzzle):
        for j in range(self.length_of_puzzle):
          print(grid[i][j],end=' ')
        print()
      print()

    def print_solution(self,state_of_grid: Grid_State):
      if state_of_grid == None:
        return
      self.print_solution(state_of_grid.parent)
      self.print_grid(state_of_grid)
```

In [15]:
```
puzzle = Sliding_Block_Grid(3)
puzzle.print_current_state_of_grid()
agent = Agent(puzzle_grid=puzzle)
```

```
1
2
3
4
5
6
0
7
8

1 2 3
4 5 6
0 7 8
```

(a) Use A* search to start from any given initial configuration and reach the goal configuration

1 2 3

4 5 6

7 8 0

with the sum of Manhattan distances of the blocks from their goal positions as the heuristic. The Manhattan distance of a pair of blocks occupying the integer n at the locations (i, p) and (j, q) (where $i, j, p, q \in \{0, \ldots, 8\}$ and $n \in \{0, \ldots, 8\}$) is given by $|i - j| + |p - q|$. Print the total number of steps taken to reach the goal and the blocks configuration at each step.

In [16]:
```
agent.astar_search_using_manhattan()
print(f'Astar search using total manhattan distance as heuristic explored
```

```
Solution found

1  2  3
4  5  6
0  7  8


1  2  3
4  5  6
7  0  8


1  2  3
4  5  6
7  8  0

Astar search using total manhattan distance as heuristic explored 3 state
s
```

(b) Repeat part (a) with the following alternative heurisitics.

1. the number of misplaced blocks
2. the Manhattan distance of the "0" block alone instead of the sum

Compare the performance of the three heuristics using the size of the explored list as the measure.

In [17]:
```
# before running this cell change the __lt__ function of GridState

agent.astar_search_using_number_of_misplaced_blocks()

print(f'Astar search using total number of misplaced tiles as heuristic e
```

```
Solution found

1 2 3
4 5 6
0 7 8


1 2 3
4 5 6
7 0 8


1 2 3
4 5 6
7 8 0

Astar search using total number of misplaced tiles as heuristic explored
3 states
```

In [18]:

```
# before running this cell change the __lt__ function of GridState



agent.astar_search_using_manhattan_distance_of_zero()
print(f'astar search using manhattan distance of zero as heuristic explor
```

```
Solution found

1 2 3
4 5 6
0 7 8


1 2 3
4 5 6
7 0 8


1 2 3
4 5 6
7 8 0

astar search using manhattan distance of zero as heuristic explored 3 sta
tes
```

In [22]:

```
# to compute efficiency
explored_astar = 0
for _ in range(100):
  puzzle_new = Sliding_Block_Grid(input_puzzle=False)
  agent_new = Agent(puzzle_new)
  agent_new.astar_search_using_manhattan(False)
  explored_astar += agent_new.astar_explore_count_manhattan_distance
  del agent_new
  del puzzle_new
print(f'the average number of vertices explored is {explored_astar/100}')
```

```
the average number of vertices explored is 568.0
```

In [23]:
```
# before running this cell change the __lt__ function of GridState

explored = 0
for _ in range(100):
  puzzle_new = Sliding_Block_Grid(input_puzzle=False)
  agent_new = Agent(puzzle_new)
  agent_new.astar_search_using_number_of_misplaced_blocks(False)
  explored += agent_new.astar_explore_count_number_of_misplaced_blocks
  del agent_new
  del puzzle_new
print(f'the average number of vertices explored is {explored/100}')
```

```
the average number of vertices explored is 964.92
```

In [24]:
```
# before running this cell change the __lt__ function of GridState

explored = 0
for i in range(100):
  puzzle_new = Sliding_Block_Grid(input_puzzle=False,depth_for_generating
  agent_new = Agent(puzzle_new)
  agent_new.astar_search_using_manhattan_distance_of_zero(False)
  explored += agent_new.astar_explore_count_manhattan_distance_of_zero
  del agent_new
  del puzzle_new
print(f'the average number of vertices explored is {explored/100}')
```

```
the average number of vertices explored is 30003.55
```

In [ ]: