**Team Members:**

**2403res81 , 2303res132, 2303res204**

**Summary of the 2403res81\_2303res132\_2303res204.py file**

**Note : Some functions mentioned in this assignment are taken from previous assignment file as similar functions could be utilized here**

1. How to run the 2403res81\_2303res132\_2303res204.py

python 2403res81\_2303res132\_2303res204.py

2. Functions used and there functionality

* 1. def create\_initial\_state(state\_values) : This function receives state\_values which is a list containing values from 1 to 8 and letter ‘B’ which signifies a blank tile in the grid . This list is used to generate a 3\*3 grid which random placements of state\_values. To generate random positions of state\_values , it is shuffled using the python package “random” function “shuffle”.

For eg : state\_values = [1,2,3,4,5,6,7,8,’B’]

After applying a random package function shuffle , it will change it to ['B', 3, 5, 4, 7, 8, 2, 6, 1] .

This output will be different when called again.

* 1. def find\_blank\_tile(state): This function receives a state matrix and returns a list of rows and columns where value ‘B’ is found.

| def find\_blank\_tile(state):  """  This function iterates on every tile and gives  the location of blank tile  :param state: state is input matrix  :return: [row,col] row and col at which blank tile  is present  """  rows = len(state)  cols = len(state[0])  for row in range(rows):  for col in range(cols):  if state[row][col] == 'B':  return [row,col] |
| --- |

* 1. def generate\_new\_states(state): This function receives a state matrix and returns a list that contains matrices with valid moves. In this function first we find a blank tile by calling function **find\_blank\_tile** and based on the returned row and col of blank tile ,new matrices are generated by iterating on valid moves.

| def generate\_new\_states(state):  """  This function generate new states based on information  present in question.  The blank can have 4 moves up,down,left and right  :param state: state is input matrix  :return: list of matrices by performing 4 moves  """  new\_states = [] # output variable to store the generated states   rows = len(state) # rows present in state matrix  cols = len(state[0]) # cols present in state matrix   # We can move either one block up,down, left or right  moves\_possible = [[-1,0],[1,0],[0,-1],[0,1]] # [up,down,left,right]   # find the blank space represented as 'B'  blank\_space = find\_blank\_tile(state) # here we get [row,col] of blank space   # iterate over all the moves and find the states  for move in moves\_possible:  new\_state = [row[:] for row in state]  new\_row = blank\_space[0]+move[0]  new\_col = blank\_space[1]+move[1]  if new\_row >= 0 and new\_row < rows and new\_col >= 0 and new\_col < cols:  #swap the value of new row and col with new legal move  temp = new\_state[new\_row][new\_col]  new\_state[new\_row][new\_col] = new\_state[blank\_space[0]][blank\_space[1]]  new\_state[blank\_space[0]][blank\_space[1]]=temp  new\_states.append(new\_state)  return new\_states |
| --- |

* 1. def convert\_arr\_tuple(matrix) : This function converts a matrix which is list of list to tuple of tuple. This function is needed so that we can easily compare arrays after pushing them into set().

| def convert\_arr\_tuple(matrix):  """  This function convert matrix to a tuple  :param arr: matrix is input state matrix  :return: tuple of tuple of rows of matrix  """  return tuple(map(tuple,matrix)) |
| --- |

* 1. def convert\_tuple\_arr(node) : This function converts a tuple of tuple to a list of list. This function is needed so that we can get the original matrix form.

| def convert\_tuple\_arr(node):  """  This function converts tuple to a matrix  :param arr: matrix is input state tuple  :return: list of list of rows of tuple  """  return list(map(list,node)) |
| --- |

* 1. def h1() : This function returns constant 0.

| def h1():  '''  This function return constant heuristic as 0  :return: constant 0  '''  return 0 |
| --- |

* 1. def h2(initial,goal): This function calculates total number of misplaced tiles and return it as cost.

| def h2(initial,goal):  '''  This function computes cost of number of misplaced tiles  :param initial: start state matrix  :param goal: target matrix  :return: cost of misplaced tiles  '''  cost = 0  for row in range(len(initial)):  for col in range(len(initial[0])):  if initial[row][col] != goal[row][col] and initial[row][col]!='B':  cost+=1  return cost |
| --- |

* 1. def h3(initial,goal): This function calculates manhattan distance between initial state and final state.

| def h3(initial,goal):  '''  This function computes the manhattan distance  :param initial:start state matrix  :param goal:target matrix  :return:total cost of manhattan distance tiles  '''  cost = 0  for row in range(len(initial)):  for col in range(len(initial[0])):  if initial[row][col] != 'B':  initial\_tile\_position = find\_tile\_location(initial,initial[row][col])  goal\_tile\_position = find\_tile\_location(goal,initial[row][col])  temp\_cost = abs(initial\_tile\_position[0] - goal\_tile\_position[0]) + \  abs(initial\_tile\_position[1]-goal\_tile\_position[1])  cost += temp\_cost  return cost |
| --- |

* 1. def h4(initial,goal): This function return 10 times the actual cost and is implemented as 10\*h3 to satisfy the condition h(n)>h\*(n)

| def h4(initial,goal):  '''  This function overestimates the actual cost  :param initial: start state matrix  :param goal:target matrix  :return: overestimated cost  '''  return 10\*h3(initial,goal) |
| --- |

* 1. def find\_tile\_location(state,tile) : This function finds the tile location in the given input matrix and return [row,column].

| def find\_tile\_location(state,tile):  '''  This function finds the location of tile in input matrix  :param state: input state matrix  :param tile: tile number ; for this eg 1,2....8  :return:  '''  for row in range(len(state)):  for col in range(len(state[0])):  if tile == state[row][col]:  return [row,col] |
| --- |

* 1. def heuristic\_map(heuristic,start,target): This function is maps to which heuristic function to use.

| def heuristic\_map(heuristic,start,target):  '''  This function maps the start and target to right heuristic function  :param heuristic: which heuristic is used 1,2,3or 4  :param initial: start state matrix  :param goal:target matrix  :return:  '''  if heuristic == 1:  return h1()  elif heuristic == 2:  return h2(start,target)  elif heuristic == 3:  return h3(start,target)  else:  return h4(start,target) |
| --- |

* 1. def check\_monotonicity(path,heuristic,target,cost): This function checks whether heuristic functions follow monotonous behaviour or not.

| def check\_monotonicity(path,heuristic,target,cost):  '''  This function checks the condition h(n) <= cost(n,m)+h(n)  :param path: optimal a\* path  :param heuristic: which heuristic function to be used.  :param target: target matrix  :param cost: cost of nodes from n to m  :return: None  '''  flag = True  start\_state = path.pop(0)  for neighbor in path:  if heuristic\_map(heuristic,convert\_tuple\_arr(start\_state),target) > \  (cost[neighbor] + heuristic\_map(heuristic,convert\_tuple\_arr(neighbor),target)):  flag = False  break  if flag:  print(f"Heuristic h({heuristic}) : Monotonicity check passed")  else:  print(f"Heuristic h({heuristic}) : Monotonicity check failed")  return |
| --- |

* 1. def check\_states\_in\_better\_included\_inferior(path1,path2) : This function verifies whether all states explored by better heuristic are also explored by inferior heuristic.

| def check\_states\_in\_better\_included\_inferior(path1,path2):  '''  This function evaluates whether states of better heuristic  are already explored by inferior heuristic  :param path1: closed list of inferior heuristic  :param path2: closed list of better heuristic  :return:  '''  flag = True  for node in path2:  if node not in path1:  flag = False  if flag:  print("Passed :All the states expanded by better heuristics should also be expanded by inferior heuristics")  else:  print("Failed :All the states expanded by better heuristics should also be expanded by inferior heuristics")  return |
| --- |

* 1. def a\_star\_algorithm​​(start, target,heuristic=1) : Complete implementation of a\* algorithm.

| def a\_star\_algorithm(start, target,heuristic=1):  '''  This function implements the a\* algorithm.  :param start: start of the search  :param target: target matrix of search  :param heuristic: which heuristic to use  :return: None  '''  start\_time = time.time()  print(f"Start time : {start\_time}")  # This function keeps track of states which are found  # and yet to be explored  open\_list = set([convert\_arr\_tuple(start)])  # This function keeps track of states which are explored  closed\_list = set([])   # This function keeps track of the cost  cost = {}   # Initialize the cost from start state  cost[convert\_arr\_tuple(start)] = 0   # parents contains an adjacency map of all nodes  parents = {}  parents[convert\_arr\_tuple(start)] = convert\_arr\_tuple(start)   # count : is used to keep track of iterations taken to reach the goal state  count = len(open\_list)  while len(open\_list) > 0:  # here n represents the current node  current\_node = None  # find a node with the lowest value of f() - evaluation function  for v in open\_list:  v\_mat = convert\_tuple\_arr(v)  if current\_node == None or cost[v] + heuristic\_map(heuristic,v\_mat,target) <= cost[current\_node] + heuristic\_map(heuristic,convert\_tuple\_arr(current\_node),target):  current\_node = v;   if current\_node == None or count==1e6:  end\_time = time.time()  print(f'Failure. Path does not exist! and terminated after trying for iterations {count}')  print("Start State : ")  print(start)  print("Target State")  print(target)  print(f"End time : {end\_time}")  print(f"Total time lapsed : {end\_time - start\_time} seconds")  return None,None   # if the current node is the stop\_node  # then we begin reconstructin the path from it to the start\_node  if current\_node == convert\_arr\_tuple(target):  print("Success")  print("Start State : ")  print(start)  print("Target State")  print(target)  print(f"Total no of iterations needed to reach goal : {count}")  print(f'Total number of steps needed to explore optimal path: {cost[current\_node]}')  reconst\_path = []   while parents[current\_node] != current\_node:  reconst\_path.append(current\_node)  current\_node = parents[current\_node]   reconst\_path.append(convert\_arr\_tuple(start))   reconst\_path.reverse()  check\_monotonicity(reconst\_path, heuristic, target,cost)  print('Path found: {}'.format(reconst\_path))  print(f'States explored : {len(closed\_list)}')  end\_time = time.time()  print(f"End time : {end\_time}")  print(f"Total time lapsed : {end\_time-start\_time} seconds")  return reconst\_path, closed\_list   # for all neighbors of the current node do  new\_states= generate\_new\_states(convert\_tuple\_arr(current\_node))  for m in new\_states:  # if the current node isn't in both open\_list and closed\_list  # add it to open\_list and note n as it's parent  m\_tuple = convert\_arr\_tuple(m)  if m\_tuple not in open\_list and m\_tuple not in closed\_list:  open\_list.add(m\_tuple)  parents[m\_tuple] = current\_node  cost[m\_tuple] = cost[current\_node] + 1   # otherwise, check if it's quicker to first visit n, then m  # and if it is, update parent data and g data  # and if the node was in the closed\_list, move it to open\_list  else:  if cost[m\_tuple] > cost[current\_node] + 1:  cost[m\_tuple] = cost[current\_node] + 1  parents[m\_tuple] = current\_node   if m\_tuple in closed\_list:  closed\_list.remove(m\_tuple)  open\_list.add(m\_tuple)   # remove n from the open\_list, and add it to closed\_list  # because all of his neighbors were inspected  open\_list.remove(current\_node)  closed\_list.add(current\_node)  count+=1 |
| --- |
|  |

* 1. def main(): This is the main function which is called every time when the .py file run.

| def main():  state = [[3,2,1],[4,5,6],[8,7,'B']]  target = [[1,2,3],[4,5,6],[7,8,'B']]  state\_values = [1,2,3,4,5,6,7,8,'B']   # First try for the given state matrix in assignment .  # If it fails generate random matrix for state  print("Solution for heuristic 1 :")  recons\_path1,closed1 = a\_star\_algorithm(state,target,1)  print("Solution for heuristic 2 :")  recons\_path2,closed2 = a\_star\_algorithm(state,target,2)  print("Solution for heuristic 3 :")  recons\_path3,closed3 = a\_star\_algorithm(state,target,3)  print("\nSolution for heuristic 4 :")  recons\_path4,closed4 = a\_star\_algorithm(state,target,4)  check\_states\_in\_better\_included\_inferior(closed3,closed4)  if not(recons\_path1 and recons\_path2 and recons\_path3 and recons\_path4):  max\_states\_to\_be\_generated = 362880  while max\_states\_to\_be\_generated:  # generate new state matrix  state = create\_initial\_state(state\_values)  print("Solution for heuristic 1 :")  recons\_path1,closed1 = a\_star\_algorithm(state,target,1)  print("Solution for heuristic 2 :")  recons\_path2,closed2 = a\_star\_algorithm(state,target,2)  print("Solution for heuristic 3 :")  recons\_path3,closed3 = a\_star\_algorithm(state,target,3)  print("\nSolution for heuristic 4 :")  recons\_path4,closed4 = a\_star\_algorithm(state,target,4)  check\_states\_in\_better\_included\_inferior(closed3,closed4)  check\_states\_in\_better\_included\_inferior(closed4, closed3)  if not(recons\_path1 and recons\_path2 and recons\_path3 and recons\_path4):  max\_states\_to\_be\_generated-=1  else:  break  if max\_states\_to\_be\_generated == 0:  print(f"No solution found after trying generated states randomly for {max\_states\_to\_be\_generated} times") |
| --- |

Note : **362880** this number is chosen because for 3\*3 grid having unique values factorial(9) unique possible combinations exist.

**Sample output for given initial state and target matrix**

Initial State matrix is :

[[3, 2, 1], [4, 5, 6], [8, 7, 'B']]

Target matrix is :

[[1, 2, 3], [4, 5, 6], [7, 8, 'B']]

| Solution for heuristic 1 : Start time : 1712455685.16436 Success Start State :  [[3, 2, 1], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']] Total no of iterations needed to reach goal : 117406 Total number of steps needed to explore optimal path: 24 Heuristic h(1) : Monotonicity check passed Path found: [((3, 2, 1), (4, 5, 'B'), (8, 7, 6)), ((3, 2, 1), (4, 'B', 5), (8, 7, 6)), ((3, 2, 1), (4, 7, 5), (8, 'B', 6)), ((3, 2, 1), (4, 7, 5), (8, 6, 'B')), ((3, 2, 1), (4, 7, 'B'), (8, 6, 5)), ((3, 2, 'B'), (4, 7, 1), (8, 6, 5)), ((3, 'B', 2), (4, 7, 1), (8, 6, 5)), (('B', 3, 2), (4, 7, 1), (8, 6, 5)), ((4, 3, 2), ('B', 7, 1), (8, 6, 5)), ((4, 3, 2), (7, 'B', 1), (8, 6, 5)), ((4, 3, 2), (7, 1, 'B'), (8, 6, 5)), ((4, 3, 'B'), (7, 1, 2), (8, 6, 5)), ((4, 'B', 3), (7, 1, 2), (8, 6, 5)), ((4, 1, 3), (7, 'B', 2), (8, 6, 5)), ((4, 1, 3), (7, 2, 'B'), (8, 6, 5)), ((4, 1, 3), (7, 2, 5), (8, 6, 'B')), ((4, 1, 3), (7, 2, 5), (8, 'B', 6)), ((4, 1, 3), (7, 2, 5), ('B', 8, 6)), ((4, 1, 3), ('B', 2, 5), (7, 8, 6)), (('B', 1, 3), (4, 2, 5), (7, 8, 6)), ((1, 'B', 3), (4, 2, 5), (7, 8, 6)), ((1, 2, 3), (4, 'B', 5), (7, 8, 6)), ((1, 2, 3), (4, 5, 'B'), (7, 8, 6)), ((1, 2, 3), (4, 5, 6), (7, 8, 'B'))] End time : 1712460843.023061 Total time lapsed : 5157.858700990677 seconds  Solution for heuristic 2 : Start time : 1712460843.048874 Success Start State :  [[3, 2, 1], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']] Total no of iterations needed to reach goal : 15530 Total number of steps needed to explore optimal path: 24 Heuristic h(2) : Monotonicity check passed Path found: [((3, 2, 1), (4, 5, 'B'), (8, 7, 6)), ((3, 2, 'B'), (4, 5, 1), (8, 7, 6)), ((3, 'B', 2), (4, 5, 1), (8, 7, 6)), (('B', 3, 2), (4, 5, 1), (8, 7, 6)), ((4, 3, 2), ('B', 5, 1), (8, 7, 6)), ((4, 3, 2), (5, 'B', 1), (8, 7, 6)), ((4, 3, 2), (5, 1, 'B'), (8, 7, 6)), ((4, 3, 'B'), (5, 1, 2), (8, 7, 6)), ((4, 'B', 3), (5, 1, 2), (8, 7, 6)), ((4, 1, 3), (5, 'B', 2), (8, 7, 6)), ((4, 1, 3), (5, 7, 2), (8, 'B', 6)), ((4, 1, 3), (5, 7, 2), ('B', 8, 6)), ((4, 1, 3), ('B', 7, 2), (5, 8, 6)), ((4, 1, 3), (7, 'B', 2), (5, 8, 6)), ((4, 1, 3), (7, 2, 'B'), (5, 8, 6)), ((4, 1, 3), (7, 2, 6), (5, 8, 'B')), ((4, 1, 3), (7, 2, 6), (5, 'B', 8)), ((4, 1, 3), (7, 2, 6), ('B', 5, 8)), ((4, 1, 3), ('B', 2, 6), (7, 5, 8)), (('B', 1, 3), (4, 2, 6), (7, 5, 8)), ((1, 'B', 3), (4, 2, 6), (7, 5, 8)), ((1, 2, 3), (4, 'B', 6), (7, 5, 8)), ((1, 2, 3), (4, 5, 6), (7, 'B', 8)), ((1, 2, 3), (4, 5, 6), (7, 8, 'B'))] End time : 1712461367.096766 Total time lapsed : 524.0478920936584 seconds  Solution for heuristic 3 : Start time : 1712461367.09922 Success Start State :  [[3, 2, 1], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']] Total no of iterations needed to reach goal : 2966 Total number of steps needed to explore optimal path: 24 Heuristic h(3) : Monotonicity check passed Path found: [((3, 2, 1), (4, 5, 6), (8, 'B', 7)), ((3, 2, 1), (4, 'B', 6), (8, 5, 7)), ((3, 'B', 1), (4, 2, 6), (8, 5, 7)), (('B', 3, 1), (4, 2, 6), (8, 5, 7)), ((4, 3, 1), ('B', 2, 6), (8, 5, 7)), ((4, 3, 1), (8, 2, 6), ('B', 5, 7)), ((4, 3, 1), (8, 2, 6), (5, 'B', 7)), ((4, 3, 1), (8, 2, 6), (5, 7, 'B')), ((4, 3, 1), (8, 2, 'B'), (5, 7, 6)), ((4, 3, 1), (8, 'B', 2), (5, 7, 6)), ((4, 'B', 1), (8, 3, 2), (5, 7, 6)), ((4, 1, 'B'), (8, 3, 2), (5, 7, 6)), ((4, 1, 2), (8, 3, 'B'), (5, 7, 6)), ((4, 1, 2), (8, 'B', 3), (5, 7, 6)), ((4, 1, 2), ('B', 8, 3), (5, 7, 6)), ((4, 1, 2), (5, 8, 3), ('B', 7, 6)), ((4, 1, 2), (5, 8, 3), (7, 'B', 6)), ((4, 1, 2), (5, 'B', 3), (7, 8, 6)), ((4, 1, 2), ('B', 5, 3), (7, 8, 6)), (('B', 1, 2), (4, 5, 3), (7, 8, 6)), ((1, 'B', 2), (4, 5, 3), (7, 8, 6)), ((1, 2, 'B'), (4, 5, 3), (7, 8, 6)), ((1, 2, 3), (4, 5, 'B'), (7, 8, 6)), ((1, 2, 3), (4, 5, 6), (7, 8, 'B'))] End time : 1712461490.1359315 Total time lapsed : 123.03671145439148 seconds  Solution for heuristic 4 : Start time : 1712461490.1364706 Success Start State :  [[3, 2, 1], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']] Total no of iterations needed to reach goal : 524 Total number of steps needed to explore optimal path: 42 Heuristic h(4) : Monotonicity check failed Path found: [((3, 2, 1), (4, 5, 'B'), (8, 7, 6)), ((3, 2, 'B'), (4, 5, 1), (8, 7, 6)), ((3, 'B', 2), (4, 5, 1), (8, 7, 6)), ((3, 5, 2), (4, 'B', 1), (8, 7, 6)), ((3, 5, 2), (4, 1, 'B'), (8, 7, 6)), ((3, 5, 2), (4, 1, 6), (8, 7, 'B')), ((3, 5, 2), (4, 1, 6), (8, 'B', 7)), ((3, 5, 2), (4, 1, 6), ('B', 8, 7)), ((3, 5, 2), ('B', 1, 6), (4, 8, 7)), ((3, 5, 2), (1, 'B', 6), (4, 8, 7)), ((3, 'B', 2), (1, 5, 6), (4, 8, 7)), (('B', 3, 2), (1, 5, 6), (4, 8, 7)), ((1, 3, 2), ('B', 5, 6), (4, 8, 7)), ((1, 3, 2), (5, 'B', 6), (4, 8, 7)), ((1, 3, 2), (5, 8, 6), (4, 'B', 7)), ((1, 3, 2), (5, 8, 6), (4, 7, 'B')), ((1, 3, 2), (5, 8, 'B'), (4, 7, 6)), ((1, 3, 2), (5, 'B', 8), (4, 7, 6)), ((1, 3, 2), ('B', 5, 8), (4, 7, 6)), ((1, 3, 2), (4, 5, 8), ('B', 7, 6)), ((1, 3, 2), (4, 5, 8), (7, 'B', 6)), ((1, 3, 2), (4, 5, 8), (7, 6, 'B')), ((1, 3, 2), (4, 5, 'B'), (7, 6, 8)), ((1, 3, 'B'), (4, 5, 2), (7, 6, 8)), ((1, 'B', 3), (4, 5, 2), (7, 6, 8)), ((1, 5, 3), (4, 'B', 2), (7, 6, 8)), ((1, 5, 3), (4, 6, 2), (7, 'B', 8)), ((1, 5, 3), (4, 6, 2), (7, 8, 'B')), ((1, 5, 3), (4, 6, 'B'), (7, 8, 2)), ((1, 5, 3), (4, 'B', 6), (7, 8, 2)), ((1, 'B', 3), (4, 5, 6), (7, 8, 2)), ((1, 3, 'B'), (4, 5, 6), (7, 8, 2)), ((1, 3, 6), (4, 5, 'B'), (7, 8, 2)), ((1, 3, 6), (4, 5, 2), (7, 8, 'B')), ((1, 3, 6), (4, 5, 2), (7, 'B', 8)), ((1, 3, 6), (4, 'B', 2), (7, 5, 8)), ((1, 3, 6), (4, 2, 'B'), (7, 5, 8)), ((1, 3, 'B'), (4, 2, 6), (7, 5, 8)), ((1, 'B', 3), (4, 2, 6), (7, 5, 8)), ((1, 2, 3), (4, 'B', 6), (7, 5, 8)), ((1, 2, 3), (4, 5, 6), (7, 'B', 8)), ((1, 2, 3), (4, 5, 6), (7, 8, 'B'))] End time : 1712461494.1988003 Total time lapsed : 4.0623297691345215 seconds Failed :All the states expanded by better heuristics should also be expanded by inferior heuristics |
| --- |

Tasks :

1. Observe and verify that better heuristics expand lesser states.

Initial State matrix is :

[[3, 2, 1], [4, 5, 6], [8, 7, 'B']]

Target matrix is :

[[1, 2, 3], [4, 5, 6], [7, 8, 'B']]

Observation:

For h1(n) : States expanded = 24

For h2(n) : States expanded = 24

For h3(n) : States expanded = 24

For h4(n) : States expanded = 42

1. Observe and verify that all the states expanded by better heuristics should also be expanded by inferior heuristics.

Applying function check\_states\_in\_better\_included\_inferior (path1,path2)

path1 : inferior heuristic

path2 : better heuristic

In path1 and path2 closed lists are passed as this list keeps track of all the explored states.

1. For h1(n) and h2(n) :

| Passed :All the states expanded by better heuristics should also be expanded by inferior heuristics |
| --- |

1. For h2(n) and h3(n) :

| Failed :All the states expanded by better heuristics should also be expanded by inferior heuristics |
| --- |

1. For h3(n) and h4(n) :

| Failed :All the states expanded by better heuristics should also be expanded by inferior heuristics |
| --- |

1. For h4(n) and h3(n) :

| Failed :All the states expanded by better heuristics should also be expanded by inferior heuristics |
| --- |

3. Observe un-reachability and provide proof.

With the initial state :

[[1,2,3],[4,5,6],[8,7,’B’]]

unreachability is observed for all the heuristics

| Failure. Path does not exist! Start State :  [[1, 2, 3], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']]  Solution for heuristic 2 : Failure. Path does not exist! Start State :  [[1, 2, 3], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']]  Solution for heuristic 3 : Failure. Path does not exist!  Start State :  [[1, 2, 3], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']]  Solution for heuristic 4 : Failure. Path does not exist!  Start State :  [[1, 2, 3], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']] |
| --- |

4. Observe and verify whether the monotone restriction is followed for the following two Heuristics:

a. Monotone restriction:h(n)<=cost(n,m)+h(m)

b. Heuristic:

i. h2(n) = number of tiles displaced from their destined position.

ii. h3(n) = sum of the Manhattan distance of each tile from the goal position.

Observations :

Heuristic h(1) : Monotonicity check passed

Heuristic h(2) : Monotonicity check passed

Heuristic h(3) : Monotonicity check passed

Heuristic h(4) : Monotonicity check failed

1. Observe and verify that if the cost of the empty tile is added (considering the empty tile as another tile), then monotonicity will be violated.

Here we can observe though blank tile is added as cost in h3(n) still it doesn’t violate the monotonicity which is correct because as observed above manhattan is admissible heuristic and it is very less likely that it will violate the monotonicity. But h4(n) violates which is true but it is not violating because of blank tile cost but due to overestimated cost. Same behavior for h4(n) was above as well.

| Solution for heuristic 3 : Start time : 1712470780.056899 Success Start State :  [[3, 2, 1], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']] Total no of iterations needed to reach goal : 3094 Total number of steps needed to explore optimal path: 24 Heuristic h(3) : Monotonicity check passed Iterations : 3094 Path found: [((3, 2, 1), (4, 5, 'B'), (8, 7, 6)), ((3, 2, 1), (4, 'B', 5), (8, 7, 6)), ((3, 2, 1), (4, 7, 5), (8, 'B', 6)), ((3, 2, 1), (4, 7, 5), (8, 6, 'B')), ((3, 2, 1), (4, 7, 'B'), (8, 6, 5)), ((3, 2, 'B'), (4, 7, 1), (8, 6, 5)), ((3, 'B', 2), (4, 7, 1), (8, 6, 5)), (('B', 3, 2), (4, 7, 1), (8, 6, 5)), ((4, 3, 2), ('B', 7, 1), (8, 6, 5)), ((4, 3, 2), (7, 'B', 1), (8, 6, 5)), ((4, 3, 2), (7, 1, 'B'), (8, 6, 5)), ((4, 3, 'B'), (7, 1, 2), (8, 6, 5)), ((4, 'B', 3), (7, 1, 2), (8, 6, 5)), ((4, 1, 3), (7, 'B', 2), (8, 6, 5)), ((4, 1, 3), (7, 2, 'B'), (8, 6, 5)), ((4, 1, 3), (7, 2, 5), (8, 6, 'B')), ((4, 1, 3), (7, 2, 5), (8, 'B', 6)), ((4, 1, 3), (7, 2, 5), ('B', 8, 6)), ((4, 1, 3), ('B', 2, 5), (7, 8, 6)), (('B', 1, 3), (4, 2, 5), (7, 8, 6)), ((1, 'B', 3), (4, 2, 5), (7, 8, 6)), ((1, 2, 3), (4, 'B', 5), (7, 8, 6)), ((1, 2, 3), (4, 5, 'B'), (7, 8, 6)), ((1, 2, 3), (4, 5, 6), (7, 8, 'B'))] End time : 1712471011.342983 Total time lapsed : 231.28608393669128 seconds  Solution for heuristic 4 : Start time : 1712471011.3435452 Success Start State :  [[3, 2, 1], [4, 5, 6], [8, 7, 'B']] Target State [[1, 2, 3], [4, 5, 6], [7, 8, 'B']] Total no of iterations needed to reach goal : 498 Total number of steps needed to explore optimal path: 42 Heuristic h(4) : Monotonicity check failed Iterations : 498 Path found: [((3, 2, 1), (4, 5, 'B'), (8, 7, 6)), ((3, 2, 'B'), (4, 5, 1), (8, 7, 6)), ((3, 'B', 2), (4, 5, 1), (8, 7, 6)), ((3, 5, 2), (4, 'B', 1), (8, 7, 6)), ((3, 5, 2), (4, 1, 'B'), (8, 7, 6)), ((3, 5, 2), (4, 1, 6), (8, 7, 'B')), ((3, 5, 2), (4, 1, 6), (8, 'B', 7)), ((3, 5, 2), (4, 1, 6), ('B', 8, 7)), ((3, 5, 2), ('B', 1, 6), (4, 8, 7)), ((3, 5, 2), (1, 'B', 6), (4, 8, 7)), ((3, 'B', 2), (1, 5, 6), (4, 8, 7)), (('B', 3, 2), (1, 5, 6), (4, 8, 7)), ((1, 3, 2), ('B', 5, 6), (4, 8, 7)), ((1, 3, 2), (5, 'B', 6), (4, 8, 7)), ((1, 3, 2), (5, 8, 6), (4, 'B', 7)), ((1, 3, 2), (5, 8, 6), (4, 7, 'B')), ((1, 3, 2), (5, 8, 'B'), (4, 7, 6)), ((1, 3, 2), (5, 'B', 8), (4, 7, 6)), ((1, 3, 2), ('B', 5, 8), (4, 7, 6)), ((1, 3, 2), (4, 5, 8), ('B', 7, 6)), ((1, 3, 2), (4, 5, 8), (7, 'B', 6)), ((1, 3, 2), (4, 5, 8), (7, 6, 'B')), ((1, 3, 2), (4, 5, 'B'), (7, 6, 8)), ((1, 3, 'B'), (4, 5, 2), (7, 6, 8)), ((1, 'B', 3), (4, 5, 2), (7, 6, 8)), ((1, 5, 3), (4, 'B', 2), (7, 6, 8)), ((1, 5, 3), (4, 6, 2), (7, 'B', 8)), ((1, 5, 3), (4, 6, 2), (7, 8, 'B')), ((1, 5, 3), (4, 6, 'B'), (7, 8, 2)), ((1, 5, 3), (4, 'B', 6), (7, 8, 2)), ((1, 'B', 3), (4, 5, 6), (7, 8, 2)), ((1, 3, 'B'), (4, 5, 6), (7, 8, 2)), ((1, 3, 6), (4, 5, 'B'), (7, 8, 2)), ((1, 3, 6), (4, 5, 2), (7, 8, 'B')), ((1, 3, 6), (4, 5, 2), (7, 'B', 8)), ((1, 3, 6), (4, 'B', 2), (7, 5, 8)), ((1, 3, 6), (4, 2, 'B'), (7, 5, 8)), ((1, 3, 'B'), (4, 2, 6), (7, 5, 8)), ((1, 'B', 3), (4, 2, 6), (7, 5, 8)), ((1, 2, 3), (4, 'B', 6), (7, 5, 8)), ((1, 2, 3), (4, 5, 6), (7, 'B', 8)), ((1, 2, 3), (4, 5, 6), (7, 8, 'B'))] End time : 1712471017.459284 Total time lapsed : 6.115738868713379 seconds Failed :All the states expanded by better heuristics should also be expanded by inferior heuristics Failed :All the states expanded by better heuristics should also be expanded by inferior heuristics |
| --- |

|  | h1(n) | h2(n) | h3(n) | h4(n) |
| --- | --- | --- | --- | --- |
| Total states explored | 117406 | 15530 | 2966 | 524 |
| Optimal  Path | 24 | 24 | 24 | 42 |
| Optimal Path Cost | 24 | 24 | 24 | 42 |
| Total Execution Time (in seconds) | 5157.858700990677 | 524.0478920936584 | 123.03671145439148 | 4.0623297691345215 seconds |

1. Compare and contrast the results of all four heuristics, h1(n), h2(n), h3(n), and h4(n), and state the reasons in a document file ‘Why one heuristic is better than the other one?’. While explaining, please comment on the optimality, time, etc.

For given initial state in assignment.

In terms of time : h4 is found to be best.

In terms of steps : h1 , h2 and h3 all give the same steps and are lesser than h4 which has less time.

Overall if we want to take a heuristic function h3 spans out to be best keeping a bias towards time + steps + motonocity+optimality all together.