Artificial Intelligence (CS561)
AI & ML Laboratory (CS571)

Assignment #2 : A* Search

Submission Date: 3rd April'2024

Group of 4 :

1. ***Saumyen Mishra (2304RES06)***
2. ***Ashish Kumar Ray (2403RES13)***
3. ***Saurav Kumar Jha (2403RES69)***
4. ***Abani Prasad (IITP002172)***

Semester-I
M.Tech AI & DSE
IIT Patna

CONTENTS

Chapter 1:
Python Code for solving the **8-Puzzle Problem** with **A-Star** Search Algorithm

<u>List of all Global-Variables & User-Defined Functions:</u>

1. **Blank_Tile_Position (matrix) :**

   This function takes a 3x3 matrix form of any 8-puzzle state as an input argument and returns the coordinates of the Blank-Tile or the Zero-Tile.
   This function was also used in the BFS-DFS Assignment.

```
#***************************************************************************************
#***************************************************************************************

def Blank_Tile_Position(matrix) :

  p=0
  q=0

  for i in range (0,3,1) :
    for j in range (0,3,1) :
      if (matrix[i][j]==0) :
        p=i
        q=j
        pos=[p,q]
  return (pos)


#***************************************************************************************
#***************************************************************************************
```

2. **GS:**

   GS or Goal State is the global variable defined/hardcoded in the python code as below:

```
GS =
[[1,2,3],
[4,5,6],
[7,8,0]]
```

3. **Any_Tile_Position (num, matrix):**
   This function takes two inputs, first one is a number between 0-8 in the 8-Puzzle Problem, and the second one is any state in the 8-Puzzle Problem. It returns the matrix-coordinates/positions of the given number in that given state of the 8-Puzzle Problem.

```python
#**********************************************************************************************
def Any_Tile_Position(num, matrix) :
  p=0
  q=0
  for i in range (0,3,1) :
    for j in range (0,3,1) :
      if (matrix[i][j]==num) :
        p=i
        q=j
        pos=[p,q]
  return (pos)

#**********************************************************************************************
```

4. **convergence_calculator (matrix):**

This empirical function is basically used to know beforehand (prior to the start of the search algorithms) whether any chosen Initial state for the 8-Puzzle problem will finally converge to the goal state or not.

<u>Purpose:</u>
So, if the chosen initial state in the 8-puzzle problem is non-converge-able, the only way for us to know from A* Search-Algorithm point of view is after many-many iterations, a lot of time and computation resources consumption later, i.e. when the OPEN_LIST or the list of unexplored items becomes empty.
This was a real problem for us, as while testing the performance of our search algorithms implementation against different initial states, when we started encountering a lot of iterations (e.g. >1,50,000 iterations), and systems hanging/slowing down drastically because of it, we found out that due to platform/HW constraints and sometimes due to unexplained program-terminations, we are not able to conclude whether the initial state chosen for the test purpose is actually nearing the goal state (which could be the evidence of our implementation working properly) or is it simply non-converge-able.

<u>Idea:</u>
Since the Goal State Matrix has all its elements arranged in an ascending order in its row-major-order form, it has no such element, in any such position, such that :

$(([i] < [j]) \cap (row\_major\_order[i] > row\_major\_order[j]))$ == True
$where : i, j \in [0,8]$

This property of the Goal State of the 8-Puzzle problem is called as having 0-Inversions.

And also, since it is a fact that any movement of the Zero/Blank Tile within the matrix, introduces (adds or subtracts) an even number of inversions; *So it basically means that if we want to start with any initial state in the 8-Puzzle State space with 'k' no.s of inversions, and keep on moving the blank tile introducing even no. of inversions at each step, to eventually reach the goal state which has 0 inversions, **"k" must always be an even number.***

Hence if no. of inversions in the chosen initial state of the 8-puzzle problem is even, that initial state will converge to the Goal State, and if it's odd, then that initial state will never converge to the Goal State.

Example : [[0,1,2],[3,4,5],[8,6,7] has 2 inversions [EVEN] and it will converge!
[[1,2,3],[4,5,6],[8,7,0]] has 1 inversion [ODD] and hence, it will not converge!

Implementation:

a.) It takes the chosen initial state for the 8-Puzzle Problem, in the form of a 3x3 square matrix. (It assumes that we treat the Blank-tile as the Zero-Tile).

b.) It then, converts the square matrix into a list of 9 numbers in row-major-order.

c.) Zero is removed from this list and after that, the number of inversions existing in the given initial state are calculated.

d.) It's checked whether the total number of inversions detected in the row-major-order form of the input initial state is even or odd.

e.) If Odd, we conclude that the Goal State GS is not reachable from the given Initial State.
We also return a flag conv=0.

f.) If Even, we conclude that the Goal State GS is reachable from the given Initial State.
We also return a flag conv=1.

```
#*******************************************************************************************

def convergence_calculator(matrix) :

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]

  row_major_order=[a[0],a[1],a[2],b[0],b[1],b[2],c[0],c[1],c[2]]
  row_major_order.remove(0)
   #print(f"\nRow  Major  Order  form  of  Input  Matrix  :  {matrix}  is  given  as  :
{row_major_order} and it's length is : {len(row_major_order)}")

  k=0

  for i in range (0,8,1) :
    for j in range (0,8,1) :
      if ((i<j) and (row_major_order[i]>row_major_order[j])) :
        k=k+1

  #print (f"\nNo. of Inversions in this given 8-Puzzle Problem is {k}")

  if (k%2 == 0) :
     print(f"\nGiven  Matrix  :  {matrix}  has  {k}  numbers  of  inversions  and  since
it's an EVEN Integer, this 8-Puzzle Problem will CONVERGE to the Goal State!")
    conv=1
  else :
     print(f"\nGiven  Matrix  :  {matrix}  has  {k}  numbers  of  inversions  and  since
it's an ODD Integer, this 8-Puzzle Problem will NOT CONVERGE to the Goal State!")
    conv=0


  return conv

#*******************************************************************************************
```

5. **h1 (matrix):**

Heuristic Function 1, takes a matrix as an input and returns 0 irrespective of the matrix, or its perceived closeness/proximity to goal state. So, h1(matrix) = 0, (and the assumption of uniform unit cost for any down, right, left or up (valid) movement of the blank-tile) essentially reduces our A* Search algorithms to behave like a Blind/Uniformed Breadth first search algorithms over the 8-Puzzle-State-Space.
However, in terms of performance comparison to the vanilla-queue BFS implementation, this A* Search version with h1=0, is dramatically slower!

It is so, because unlike Vanilla BFS, in every iteration, we don't just blindly pop out the last element from the queue of undiscovered elements in A* Search. Here, we need to traverse the Open-List in every iteration, find the element with the minimum forward + backward cost, and pop that element of the list. So, since we are not gaining any useful domain knowledge from **h1**, it is not able to guide our search algorithm any better, and also makes it slower due to additional traversal and calculation costs over the open-lis
.
But, still, by definition, since the value of this heuristic function does not increase (as it always stays 0) over the optimal-path, it's an admissible and monotonic heuristic.

```
#*******************************************************************************************

def h1(matrix) :

  return 0

#*******************************************************************************************
```

6. **h2 (matrix):**

Heuristic Function 2, takes a matrix as an input and returns the numbers of misplaced tiles w.r.t goal state and hence, it's a relative measure of the perceived closeness/proximity to goal state. We do not take into account the position of the zero/blank tile while making this calculation. It is also a consistent, monotonic, non-increasing over optimal-path and admissible heuristic. Since it gives us a better perspective on the perceived proximity of a state to the goal state, it helps A* by expanding/exploring much lesser intermediary nodes as compared to h1.

```
#*******************************************************************************************
def h2(matrix) :

  misplaced_tiles=0

  for i in range(0,3,1) :
    for j in range(0,3,1) :

      if ((i==0) and (matrix[i][j]!=(i+j+1))) :
        misplaced_tiles=misplaced_tiles+1

      elif ((i==1) and (matrix[i][j]!=(i+j+3))) :
        misplaced_tiles=misplaced_tiles+1

      elif ((i==2) and (matrix[i][j]!=(i+j+5))) :
        misplaced_tiles=misplaced_tiles+1
  #print(f"\nNumber of misplaced tiles is {misplaced_tiles-1}")
```

```
  misplaced_tiles=misplaced_tiles-1
  return misplaced_tiles

#*******************************************************************************************
```

7. **h3 (matrix):**

Heuristic Function 3, takes a matrix as an input and returns the sum of the Manhattan distances w.r.t goal state and hence, it's even a better relative measure of the perceived closeness/proximity to goal state. We do not take into account the position of the zero/blank tile while making this calculation. It is also a consistent, monotonic, and non-increasing over optimal-path, and an admissible heuristic. Since it gives us a better perspective on the perceived proximity of a state to the goal state (even better, compared to h2), it helps A* by expanding/exploring much least number of intermediary nodes if compared to h1 or h2.

```
#*******************************************************************************************
def h3(matrix) :

  manhattan_distance=0


  for i in range (1,9,1) :
    pos1=Any_Tile_Position(i, matrix)
    pos2=Any_Tile_Position(i, GS)

    cartesian_euclidean_dist=math.sqrt(((pos1[0]-pos2[0])**2) + ((pos1[1]-pos2[1])**2))

    if (int(cartesian_euclidean_dist)==0) :
      continue
    elif ((cartesian_euclidean_dist)%1==0) :
      manhattan_distance = manhattan_distance + int(cartesian_euclidean_dist)
    elif ((cartesian_euclidean_dist)==math.sqrt(8)) :
      p=0
      p=int(cartesian_euclidean_dist)+2
      manhattan_distance = manhattan_distance + p
    else :
      k=0
      k=int(cartesian_euclidean_dist)+1
      manhattan_distance = manhattan_distance + k

  return manhattan_distance
#*******************************************************************************************
```

8. **h4 (matrix):**

Heuristic Function 4, takes a matrix as an input and returns its relative position squared weightage as a measure of perceived proximity/closeness to the Goal State. However, it's not a consistent or admissible heuristic, as it's defined to behave in a particular way (consistently) in one partition of the state space and behave differently (inconsistently) in another partition of the state space.

```
#*****************************************************************************************
def h4(matrix) :

  position_squared_weightage=0

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]

  row_major_order=[a[0],a[1],a[2],b[0],b[1],b[2],c[0],c[1],c[2]]
  row_major_order.remove(0)

  k=0

  for i in range (0,8,1) :
    for j in range (0,8,1) :
      if ((i<j) and (row_major_order[i]>row_major_order[j])) : #[12835674]
        k=k+1

  if k<=10 :
    for i in range (0, 8, 1) :
                           position_squared_weightage=position_squared_weightage        +
(row_major_order[i]*((i+1)**2))

    position_squared_weightage_relative = 1296 - position_squared_weightage

  if k>10 :
    for i in range (0, 8, 1) :
                           position_squared_weightage=position_squared_weightage        +
((row_major_order[i]**2)*((i+1)**2))

    position_squared_weightage_relative = 9999 - position_squared_weightage

  return position_squared_weightage_relative
#*****************************************************************************************
```

9. **Move_Generator (matrix):**

This function takes a 3-by-3 matrix (Matrix form of the 8-Puzzle Problem) as an argument and returns a list which is a set of all possible neighbors/moves from the current state (i.e. input argument matrix). As mentioned above it also needs input from the user-defined function 'Blank_Tile_Position (matrix)' to locate the zero-tile. Then, it finds the set of possible neighbors by:

- Swapping the positions of zero-tile & up-element (If Zero-Tile is in rows [1] or [2] of matrix) OR,
- Swapping the positions of zero-tile & down-element or, (If Zero-Tile is in rows [0] or [1] of matrix) OR,

- Swapping the positions of zero-tile & left-element or, (If Zero-Tile is in columns [1] or [2] of matrix) OR,
- Swapping the positions of zero-tile & right-element or, (If Zero-Tile is in columns [0] or [1] of matrix)

```
#***********************************************************************************
def Move_Generator(matrix) :

  neighbour_set=[]
  a2=Blank_Tile_Position(matrix)

  if (matrix==[[1,2,3],[4,5,6],[7,8,0]]) :
    neighbour_set = []
    print (f"\nDetected in Move_Generator() : Goal State is reached!")
    print (matrix)

  else :

    s1=matrix[0][:]
    s2=matrix[1][:]
    s3=matrix[2][:]
    Neighbour_Down=[s1,s2,s3]

    ns1=matrix[0][:]
    ns2=matrix[1][:]
    ns3=matrix[2][:]
    Neighbour_Left=[ns1,ns2,ns3]

    ss1=matrix[0][:]
    ss2=matrix[1][:]
    ss3=matrix[2][:]
    Neighbour_Right=[ss1,ss2,ss3]

    nss1=matrix[0][:]
    nss2=matrix[1][:]
    nss3=matrix[2][:]
    Neighbour_Up=[nss1,nss2,nss3]

    if (a2==[0,0]) :
      n_down=Neighbour_Down
      n_right=Neighbour_Right

      temp_down = n_down[1][0]
      n_down[1][0]=0
      n_down[0][0]=temp_down

      temp_right = n_right[0][1]
      n_right[0][1]=0
      n_right[0][0]=temp_right

      neighbour_set=[n_down,n_right]

    elif(a2==[0,1]) :
      n_down=Neighbour_Down
      n_right=Neighbour_Right
      n_left=Neighbour_Left
```

```
    temp_down = n_down[1][1]
    n_down[1][1]=0
    n_down[0][1]=temp_down

    temp_right = n_right[0][2]
    n_right[0][2]=0
    n_right[0][1]=temp_right

    temp_left = n_left[0][0]
    n_left[0][0]=0
    n_left[0][1]=temp_left

    neighbour_set=[n_down,n_right,n_left]

  elif(a2==[0,2]) :
    n_down=Neighbour_Down
    n_left=Neighbour_Left

    temp_down = n_down[1][2]
    n_down[1][2]=0
    n_down[0][2]=temp_down

    temp_left = n_left[0][1]
    n_left[0][1]=0
    n_left[0][2]=temp_left

    neighbour_set=[n_down,n_left]

  elif(a2==[1,0]) :
    n_down=Neighbour_Down
    n_right=Neighbour_Right
    n_up=Neighbour_Up

    temp_down = n_down[2][0]
    n_down[2][0]=0
    n_down[1][0]=temp_down

    temp_right = n_right[1][1]
    n_right[1][1]=0
    n_right[1][0]=temp_right

    temp_up = n_up[0][0]
    n_up[0][0]=0
    n_up[1][0]=temp_up

    neighbour_set=[n_down,n_right,n_up]

  elif(a2==[1,1]) :
    n_down=Neighbour_Down
    n_right=Neighbour_Right
    n_left=Neighbour_Left
    n_up=Neighbour_Up

    temp_down = n_down[2][1]
```

```
    n_down[2][1]=0
    n_down[1][1]=temp_down

    temp_right = n_right[1][2]
    n_right[1][2]=0
    n_right[1][1]=temp_right

    temp_left = n_left[1][0]
    n_left[1][0]=0
    n_left[1][1]=temp_left

    temp_up = n_up[0][1]
    n_up[0][1]=0
    n_up[1][1]=temp_up

    neighbour_set=[n_down,n_right,n_left,n_up]

  elif(a2==[1,2]) :
    n_down=Neighbour_Down
    n_left=Neighbour_Left
    n_up=Neighbour_Up

    temp_down = n_down[2][2]
    n_down[2][2]=0
    n_down[1][2]=temp_down

    temp_left = n_left[1][1]
    n_left[1][1]=0
    n_left[1][2]=temp_left

    temp_up = n_up[0][2]
    n_up[0][2]=0
    n_up[1][2]=temp_up

    neighbour_set=[n_down,n_left,n_up]

  if (a2==[2,0]) :
    n_right=Neighbour_Right
    n_up=Neighbour_Up

    temp_right = n_right[2][1]
    n_right[2][1]=0
    n_right[2][0]=temp_right

    temp_up = n_up[1][0]
    n_up[1][0]=0
    n_up[2][0]=temp_up

    neighbour_set=[n_right,n_up]

  elif(a2==[2,1]) :
    n_right=Neighbour_Right
    n_left=Neighbour_Left
    n_up=Neighbour_Up
```

```
      temp_right = n_right[2][2]
      n_right[2][2]=0
      n_right[2][1]=temp_right

      temp_left = n_left[2][0]
      n_left[2][0]=0
      n_left[2][1]=temp_left

      temp_up = n_up[1][1]
      n_up[1][1]=0
      n_up[2][1]=temp_up

      neighbour_set=[n_right,n_left,n_up]

    elif(a2==[2,2]) :
      n_left=Neighbour_Left
      n_up=Neighbour_Up

      temp_left = n_left[2][1]
      n_left[2][1]=0
      n_left[2][2]=temp_left

      temp_up = n_up[1][2]
      n_up[1][2]=0
      n_up[2][2]=temp_up
      neighbour_set=[n_left,n_up]

  return neighbour_set

#*********************************************************************************
```

10. heuristics_monotonicity_checker (lis,num)

This function takes two arguments:

- **lis:** List of all nodes in the optimal path from the start state to the goal state. This is constructed by the A* Search algorithm itself once the goal state is reached!
- **num:** Heuristic ID (e.g. : h1=0, h2:No. of misplaced tiles, h3: Sum of Manhattan Distances, h4: Relative Position Squared Weightage)

It evaluates the elements in the optimal path from start state to goal state and calculates the heuristics value corresponding to each of those elements. (Which heuristic to be used depends upon which A* variant we have deployed!)
It then checks for monotonicity/non-increasing/consistency property as it checks whether all heuristics are also arranged in a non-increasing order along that optimal path.

```
#*********************************************************************************
def heuristics_monotonicity_checker(lis,num) :

  #lis is the 'path' output from the A* Search Algorithm!

  monotonicity_flag=0
```

```
  if (num==3) :
    int_3=[]
    for p3 in range (0, len(lis), 1) :
      int_3.append(h3(lis[p3]))

    k1=int_3
    #print(int_2)
    k1.sort(reverse=True)
    int_3_sorted = k1

    monotonicity_flag = (int_3_sorted==int_3)
    if monotonicity_flag :
       print ("\nh3(n) or Sum of Manhattan distances is a consistent & monotonic
Heuristic!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")



  elif (num==2) :
    int_2=[]
    for p2 in range (0, len(lis), 1) :
      int_2.append(h2(lis[p2]))

    k=int_2
    #print(int_2)
    k.sort(reverse=True)
    int_2_sorted = k

    #print(int_2_sorted)
    monotonicity_flag = (int_2_sorted==int_2)
    if monotonicity_flag :
       print ("\nh2(n) or Numbers of Misplaced Tiles is a consistent & monotonic
Heuristic!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")


  elif (num==1) :
    int_1=[]
    for p1 in range (0, len(lis), 1) :
      int_1.append(h1(lis[p1]))

    k2=int_1
    #print(int_2)
    k2.sort(reverse=True)
    int_1_sorted = k2

    #print(int_2_sorted)
    monotonicity_flag = (int_1_sorted==int_1)
    if monotonicity_flag :
      print ("\nh1(n)=0 a consistent & admissible Heuristic!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")
```

```
  elif (num==4) :
    int_4=[]
    for p4 in range (0, len(lis), 1) :
      int_4.append(h4(lis[p4]))
    #print(f"\nInt_4 : {int_4}\n")

    k4=int_4
    #print(int_2)
    k4.sort(reverse=True)
    int_4_sorted = k4

    #print(f"\nInt_4_Sorted : {int_4_sorted}\n")

    #print(int_2_sorted)
    monotonicity_flag = (int_4_sorted==int_4)
    if monotonicity_flag :
        print ("\nh4(n) or, Position-Squared-Weightage-Relative a consistent & admissible Heuristic for the given initial state!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")

  return monotonicity_flag
#*********************************************************************************************
```

11. explored_states_inclusivity_checker (LIST_1, LIST_2)

This function takes two lists as inputs:

- LIST_1: CLOSED_LIST Output from A* Algorithm using the better heuristic, after convergence has reached!
- LIST_2: CLOSED_LIST Output from A* Algorithm using the inferior heuristic, after convergence has reached!

It then checks taking each element in LIST_1 and searching for it inside LIST_2.
If all elements expanded/explored by better heuristics are contained within the closed_list of the inferior heuristics, we will say that inclusivity check is passed for this pair of heuristics.

```
#############################################################################################

def explored_states_inclusivity_checker (LIST_1,LIST_2): #LIST_1 is closed list of superior heuristic, LIST_2 is closed list of inferior heuristic
  clist_elements_1 = []
  clist_elements_2 = []

  for k1 in range(0, len(LIST_1), 1):
    clist_elements_1.append(LIST_1[k1][0])
  for k2 in range(0, len(LIST_2), 1):
    clist_elements_2.append(LIST_2[k2][0])

  inclusivity_counter=0

  for element in clist_elements_1 :
    if element in clist_elements_2:
      inclusivity_counter=inclusivity_counter+1
```

```
    #else :
      #print(element)
      #indx=clist_elements_1.index(element)
      #print(indx)

  #print (len(clist_elements_1))

  #print (len(clist_elements_2))

  #print (inclusivity_counter)

  if inclusivity_counter==len(LIST_1) :
     print("\n\nAll states explored by Superior Heuristics are also expanded by
Inferior heuristics\n\n")
  else :
    print ("\n\nInclusivity Check Failed!!!")

##################################################################################
```

12. **Complete A* Python-Code using h1, h2, h3 & h4 :**

```
#from collections import deque
#import matplotlib.pyplot as plt
import time
import tracemalloc
import math

GS=[[1,2,3],[4,5,6],[7,8,0]]

#********************************************************************************
*********************************************************************************
***********
#********************************************************************************
*********************************************************************************
***********

def Blank_Tile_Position(matrix) :

  p=0
  q=0

  for i in range (0,3,1) :
    for j in range (0,3,1) :
      if (matrix[i][j]==0) :
        p=i
        q=j
        pos=[p,q]
  return (pos)


#********************************************************************************
*********************************************************************************
***********
```

```
#*****************************************************************************************

def Any_Tile_Position(num, matrix) :

  p=0
  q=0

  for i in range (0,3,1) :
    for j in range (0,3,1) :
      if (matrix[i][j]==num) :
        p=i
        q=j
        pos=[p,q]
  return (pos)

#*****************************************************************************************
#*****************************************************************************************

def convergence_calculator(matrix) :

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]

  row_major_order=[a[0],a[1],a[2],b[0],b[1],b[2],c[0],c[1],c[2]]
  row_major_order.remove(0)
   #print(f"\nRow Major Order form of Input Matrix : {matrix} is given as : {row_major_order} and it's length is : {len(row_major_order)}")

  k=0

  for i in range (0,8,1) :
    for j in range (0,8,1) :
      if ((i<j) and (row_major_order[i]>row_major_order[j])) :
        k=k+1

  #print (f"\nNo. of Inversions in this given 8-Puzzle Problem is {k}")

  if (k%2 == 0) :
     print(f"\nGiven Matrix : {matrix} has {k} numbers of inversions and since it's an EVEN Integer, this 8-Puzzle Problem will CONVERGE to the Goal State!")
    conv=1
  else :
     print(f"\nGiven Matrix : {matrix} has {k} numbers of inversions and since it's an ODD Integer, this 8-Puzzle Problem will NOT CONVERGE to the Goal State!")
    conv=0


  return conv
```

```
#********************************************************************************
*********************************************************************************
***********
#********************************************************************************
*********************************************************************************
***********

def h1(matrix) :

  return 0

#********************************************************************************
*********************************************************************************
***********
#********************************************************************************
*********************************************************************************
***********

def h2(matrix) :

  misplaced_tiles=0

  for i in range(0,3,1) :
    for j in range(0,3,1) :

      if ((i==0) and (matrix[i][j]!=(i+j+1))) :
        misplaced_tiles=misplaced_tiles+1

      elif ((i==1) and (matrix[i][j]!=(i+j+3))) :
        misplaced_tiles=misplaced_tiles+1

      elif ((i==2) and (matrix[i][j]!=(i+j+5))) :
        misplaced_tiles=misplaced_tiles+1
  #print(f"\nNumber of misplaced tiles is {misplaced_tiles-1}")
  misplaced_tiles=misplaced_tiles-1
  return misplaced_tiles


#********************************************************************************
*********************************************************************************
***********
#********************************************************************************
*********************************************************************************
***********

def h3(matrix) :

  manhattan_distance=0


  for i in range (1,9,1) :
    pos1=Any_Tile_Position(i, matrix)
    pos2=Any_Tile_Position(i, GS)
```

```
        cartesian_euclidean_dist=math.sqrt(((pos1[0]-pos2[0])**2)   +   ((pos1[1]-
pos2[1])**2))

    if (int(cartesian_euclidean_dist)==0) :
      continue
    elif ((cartesian_euclidean_dist)%1==0) :
      manhattan_distance = manhattan_distance + int(cartesian_euclidean_dist)
    elif ((cartesian_euclidean_dist)==math.sqrt(8)) :
      p=0
      p=int(cartesian_euclidean_dist)+2
      manhattan_distance = manhattan_distance + p
    else :
      k=0
      k=int(cartesian_euclidean_dist)+1
      manhattan_distance = manhattan_distance + k

  return manhattan_distance


#*********************************************************************************
*********************************************************************************
***********
#*********************************************************************************
*********************************************************************************
***********

def h4(matrix) :

  position_squared_weightage=0

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]

  row_major_order=[a[0],a[1],a[2],b[0],b[1],b[2],c[0],c[1],c[2]]
  row_major_order.remove(0)

  k=0

  for i in range (0,8,1) :
    for j in range (0,8,1) :
      if ((i<j) and (row_major_order[i]>row_major_order[j])) : #[12835674]
        k=k+1

  if k<=10 :
    for i in range (0, 8, 1) :
                    position_squared_weightage=position_squared_weightage      +
(row_major_order[i]*((i+1)**2))

    position_squared_weightage_relative = 1296 - position_squared_weightage

  if k>10 :
    for i in range (0, 8, 1) :
                    position_squared_weightage=position_squared_weightage      +
((row_major_order[i]**2)*((i+1)**2))
```

```
    position_squared_weightage_relative = 9999 - position_squared_weightage

  return position_squared_weightage_relative



#**********************************************************************************
*****************************************
#**********************************************************************************
*****************************************

def Move_Generator(matrix) :

  neighbour_set=[]
  a2=Blank_Tile_Position(matrix)

  if (matrix==[[1,2,3],[4,5,6],[7,8,0]]) :
    neighbour_set = []
    print (f"\nDetected in Move_Generator() : Goal State is reached!")
    print (matrix)

  else :

    s1=matrix[0][:]
    s2=matrix[1][:]
    s3=matrix[2][:]
    Neighbour_Down=[s1,s2,s3]

    ns1=matrix[0][:]
    ns2=matrix[1][:]
    ns3=matrix[2][:]
    Neighbour_Left=[ns1,ns2,ns3]

    ss1=matrix[0][:]
    ss2=matrix[1][:]
    ss3=matrix[2][:]
    Neighbour_Right=[ss1,ss2,ss3]

    nss1=matrix[0][:]
    nss2=matrix[1][:]
    nss3=matrix[2][:]
    Neighbour_Up=[nss1,nss2,nss3]

    if (a2==[0,0]) :
      n_down=Neighbour_Down
      n_right=Neighbour_Right

      temp_down = n_down[1][0]
      n_down[1][0]=0
      n_down[0][0]=temp_down

      temp_right = n_right[0][1]
      n_right[0][1]=0
      n_right[0][0]=temp_right
```

```
    neighbour_set=[n_down,n_right]

  elif(a2==[0,1]) :
    n_down=Neighbour_Down
    n_right=Neighbour_Right
    n_left=Neighbour_Left

    temp_down = n_down[1][1]
    n_down[1][1]=0
    n_down[0][1]=temp_down

    temp_right = n_right[0][2]
    n_right[0][2]=0
    n_right[0][1]=temp_right

    temp_left = n_left[0][0]
    n_left[0][0]=0
    n_left[0][1]=temp_left

    neighbour_set=[n_down,n_right,n_left]

  elif(a2==[0,2]) :
    n_down=Neighbour_Down
    n_left=Neighbour_Left

    temp_down = n_down[1][2]
    n_down[1][2]=0
    n_down[0][2]=temp_down

    temp_left = n_left[0][1]
    n_left[0][1]=0
    n_left[0][2]=temp_left

    neighbour_set=[n_down,n_left]

  elif(a2==[1,0]) :
    n_down=Neighbour_Down
    n_right=Neighbour_Right
    n_up=Neighbour_Up

    temp_down = n_down[2][0]
    n_down[2][0]=0
    n_down[1][0]=temp_down

    temp_right = n_right[1][1]
    n_right[1][1]=0
    n_right[1][0]=temp_right

    temp_up = n_up[0][0]
    n_up[0][0]=0
    n_up[1][0]=temp_up

    neighbour_set=[n_down,n_right,n_up]
```

```
    elif(a2==[1,1]) :
      n_down=Neighbour_Down
      n_right=Neighbour_Right
      n_left=Neighbour_Left
      n_up=Neighbour_Up

      temp_down = n_down[2][1]
      n_down[2][1]=0
      n_down[1][1]=temp_down

      temp_right = n_right[1][2]
      n_right[1][2]=0
      n_right[1][1]=temp_right

      temp_left = n_left[1][0]
      n_left[1][0]=0
      n_left[1][1]=temp_left

      temp_up = n_up[0][1]
      n_up[0][1]=0
      n_up[1][1]=temp_up

      neighbour_set=[n_down,n_right,n_left,n_up]

    elif(a2==[1,2]) :
      n_down=Neighbour_Down
      n_left=Neighbour_Left
      n_up=Neighbour_Up

      temp_down = n_down[2][2]
      n_down[2][2]=0
      n_down[1][2]=temp_down

      temp_left = n_left[1][1]
      n_left[1][1]=0
      n_left[1][2]=temp_left

      temp_up = n_up[0][2]
      n_up[0][2]=0
      n_up[1][2]=temp_up

      neighbour_set=[n_down,n_left,n_up]

    if (a2==[2,0]) :
      n_right=Neighbour_Right
      n_up=Neighbour_Up

      temp_right = n_right[2][1]
      n_right[2][1]=0
      n_right[2][0]=temp_right

      temp_up = n_up[1][0]
      n_up[1][0]=0
      n_up[2][0]=temp_up
```

```
      neighbour_set=[n_right,n_up]

    elif(a2==[2,1]) :
      n_right=Neighbour_Right
      n_left=Neighbour_Left
      n_up=Neighbour_Up

      temp_right = n_right[2][2]
      n_right[2][2]=0
      n_right[2][1]=temp_right

      temp_left = n_left[2][0]
      n_left[2][0]=0
      n_left[2][1]=temp_left

      temp_up = n_up[1][1]
      n_up[1][1]=0
      n_up[2][1]=temp_up

      neighbour_set=[n_right,n_left,n_up]

    elif(a2==[2,2]) :
      n_left=Neighbour_Left
      n_up=Neighbour_Up

      temp_left = n_left[2][1]
      n_left[2][1]=0
      n_left[2][2]=temp_left

      temp_up = n_up[1][2]
      n_up[1][2]=0
      n_up[2][2]=temp_up
      neighbour_set=[n_left,n_up]

  return neighbour_set

#*******************************************************************************
*****************************************
#*******************************************************************************
*****************************************

def heuristics_monotonicity_checker(lis,num) :

  #lis is the 'path' output from the A* Search Algorithm!

  monotonicity_flag=0

  if (num==3) :
    int_3=[]
    for p3 in range (0, len(lis), 1) :
      int_3.append(h3(lis[p3]))

    k1=int_3
    #print(int_2)
    k1.sort(reverse=True)
```

```
    int_3_sorted = k1

    monotonicity_flag = (int_3_sorted==int_3)
    if monotonicity_flag :
      print ("\nh3(n) or Sum of Manhattan distances is a consistent & monotonic
Heuristic!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")



  elif (num==2) :
    int_2=[]
    for p2 in range (0, len(lis), 1) :
      int_2.append(h2(lis[p2]))

    k=int_2
    #print(int_2)
    k.sort(reverse=True)
    int_2_sorted = k

    #print(int_2_sorted)
    monotonicity_flag = (int_2_sorted==int_2)
    if monotonicity_flag :
      print ("\nh2(n) or Numbers of Misplaced Tiles is a consistent & monotonic
Heuristic!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")


  elif (num==1) :
    int_1=[]
    for p1 in range (0, len(lis), 1) :
      int_1.append(h1(lis[p1]))

    k2=int_1
    #print(int_2)
    k2.sort(reverse=True)
    int_1_sorted = k2

    #print(int_2_sorted)
    monotonicity_flag = (int_1_sorted==int_1)
    if monotonicity_flag :
      print ("\nh1(n)=0 a consistent & admissible Heuristic!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")

  elif (num==4) :
    int_4=[]
    for p4 in range (0, len(lis), 1) :
      int_4.append(h4(lis[p4]))
    #print(f"\nInt_4 : {int_4}\n")

    k4=int_4
    #print(int_2)
```

```
    k4.sort(reverse=True)
    int_4_sorted = k4

    #print(f"\nInt_4_Sorted : {int_4_sorted}\n")

    #print(int_2_sorted)
    monotonicity_flag = (int_4_sorted==int_4)
    if monotonicity_flag :
        print  ("\nh4(n)  or,  Position-Squared-Weightage-Relative  a  consistent  &
admissible Heuristic for the given initial state!\n")
    else :
      print("\nMONOTONICITY CHECK FAILURE\n")

  return monotonicity_flag


#*********************************************************************************
*****************************************
#*********************************************************************************
*****************************************

def astar_h1(matrix) :

  conv=convergence_calculator(matrix)

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]
  root=[a,b,c]

  clist=[]
  olist=[(root,0)]
  CP_list=[(root,"START")]
  I=0
  parent=[]


  while olist :

    I=I+1



#print("\n\n**********************************************************************
**********************************************************************************
************************************************************************")

#print("**************************************************************************
**********************************************************************************
*******************************************************************\n")

    print(f"\nIteration Number :{I}\n")


#print("\n************************************************************************
```

```
*********************************************************************************
*********************************************************************")

#print("**************************************************************************
*********************************************************************************
*******************************************************************\n")

       #print(f"\n\nFor  Iteration  No.  {I}:\n\nMembers  of  Open-List  are  :
\n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

    heur=[]
    list_int=([],0)
    list_decider=[]

    for i in range (0, len(olist), 1) :
      pot_elmt, depth_pot_elmt = olist[i]
      score = h1(pot_elmt)
      decider = score + depth_pot_elmt
      list_int = (pot_elmt,decider)
      heur.append(list_int)

     #print(f"\nIn  Iteration  Number  {I},  Members(n)  of  the  Open-List  and  their
corresponding [g(n) + h2(n)] scores : {heur}\n")

    for i1 in range(0, len(heur), 1) :
      pot_decider = heur[i1][1]
      list_decider.append(pot_decider)

    min_decider=min(list_decider)
    #print(f"\nMinimum value of [g(n) + h2(n)] Score : {min_decider}\n")
    min_decider_ind=list_decider.index(min_decider)
     #print(f"\nAND  it  is  appearing  at  index  '{min_decider_ind}'  of  the  Open
List\n")

    popped, steps = olist[min_decider_ind]
    clist.append(olist[min_decider_ind])
    olist.remove(olist[min_decider_ind])

    if (I==2000 and popped!=GS and conv==0) :
        print("\n\nFAILURE  MESSAGE  !!!\n\nSince  we  know  beforehand  from  the
convergence-calculator function,  that the given initial State will not converge,
we  are  exiting  the  A*  Algorithm  !!!\n\nThis  is  done  due  to  hardware  constraints
as  the  compilation  platform  is  unable  to  keep  computing  until  Open_List=[]
!!!\n\n")
      print("\n")
      return -1, I, -1, len(clist), olist, clist
      break

    if (popped==[[1,2,3],[4,5,6],[7,8,0]]) :

#print("\n************************************************************************
*********************************************************************************
*********************************************************************")

#print("**************************************************************************
```

```
*********************************************************************************
*******************************************************************\n")
           #print(f"\n\nFor  Iteration  No.  {I}:\n\nMembers  of  Open-List  are  :
\n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

         print(f"\n\nSUCCESS MESSAGE!!!\n\nSTART STATE: {matrix}\n\nGOAL STATE :
{GS}\n\nWITH A* ALGORITHM, GOAL STATE IS REACHED IN ITERATION No. :'{I}' AND IT
APPEARS AFTER '{steps}' STEPS FROM INITIAL STATE, \n")

        Num_Explored_Nodes= len(clist)
            print(f"\nTotal  Number  of  nodes  explored  to  reach  goal  state  :
{Num_Explored_Nodes}\n")

        #print(f"\n{CP_list}\n")

        def index_finder(element,lis) :
          i_element=0
          for x in range(len(lis)-1,0,-1) :
            if (CP_list[x][0]==element) :
              i_element=x
              break
          return i_element

        t=index_finder(GS,CP_list)
        while t:
          q1=CP_list[t][1]
          parent.append(q1)
          t=index_finder(q1,CP_list)
        parent.insert(0,GS)

        path=parent[::-1]

        print(f"\nTotal Number of States in the Optimal path : {len(path)}\n")
         print("\nIn A* Algorithm using h(n)=0 as a heuristic , the path to goal
state looks like this :\n")

        for y in range(0, len(path)-1,1) :
          print(f"\n{path[y]}==>\n")
        print(f"\n{GS}\n")

        heuristics_monotonicity_checker(path,1)



#print("\n***********************************************************************
*********************************************************************************
*********************************************************************")

#print("*************************************************************************
*********************************************************************************
*******************************************************************\n")

        return steps, I, path, Num_Explored_Nodes, olist, clist
        break
```

```
    neighbours=Move_Generator(popped)
     #print(f"\nFor Iteration {I}, and Popped State : {popped}, set of possible 
neighbours : {neighbours}\n")

    olist_elements=[]
    clist_elements=[]

    for k in range (0, len(olist), 1) :
      olist_elements.append(olist[k][0])

    for k1 in range (0, len(clist), 1) :
      clist_elements.append(clist[k1][0])

    for neighbour in neighbours:
             if  ((neighbour  not  in  olist_elements)  and  (neighbour  not  in 
clist_elements)) :
            olist.append((neighbour, steps+1))
            CP_list.append((neighbour,popped))



#********************************************************************************
*****************************************
#********************************************************************************
*****************************************

def astar_h2(matrix) :

  conv=convergence_calculator(matrix)

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]
  root=[a,b,c]

  clist=[]
  olist=[(root,0)]
  CP_list=[(root,"START")]
  I=0
  parent=[]


  while olist :

    I=I+1



#print("\n\n*********************************************************************
*********************************************************************************
***********************************************************************")

#print("*************************************************************************
*********************************************************************************
*******************************************************************\n")
```

```
    print(f"\nIteration Number :{I}\n")


#print("\n*******************************************************************************************************************************************************************************************************")

#print("******************************************************************************************************************************************************************************************************\n")

       #print(f"\n\nFor Iteration No. {I}:\n\nMembers of Open-List are : \n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

    heur=[]
    list_int=([],0)
    list_decider=[]

    for i in range (0, len(olist), 1) :
      pot_elmt, depth_pot_elmt = olist[i]
      score = h2(pot_elmt)
      decider = score + depth_pot_elmt
      list_int = (pot_elmt,decider)
      heur.append(list_int)

     #print(f"\nIn Iteration Number {I}, Members(n) of the Open-List and their corresponding [g(n) + h2(n)] scores : {heur}\n")

    for i1 in range(0, len(heur), 1) :
      pot_decider = heur[i1][1]
      list_decider.append(pot_decider)

    min_decider=min(list_decider)
    #print(f"\nMinimum value of [g(n) + h2(n)] Score : {min_decider}\n")
    min_decider_ind=list_decider.index(min_decider)
     #print(f"\nAND it is appearing at index '{min_decider_ind}' of the Open List\n")

    popped, steps = olist[min_decider_ind]
    clist.append(olist[min_decider_ind])
    olist.remove(olist[min_decider_ind])

    if (I==2000 and popped!=GS and conv==0) :
        print("\n\nFAILURE MESSAGE !!!\n\nSince we know beforehand from the convergence-calculator function, that the given initial State will not converge, we are exiting the A* Algorithm !!!\n\nThis is done due to hardware constraints as the compilation platform is unable to keep computing until Open_List=[] !!!\n\n")
      print("\n")
      return -1, I, -1, len(clist), olist, clist
      break

    if (popped==[[1,2,3],[4,5,6],[7,8,0]]) :
```

```
#print("\n***********************************************************************
*********************************************************************************
*********************************************************************")

#print("*************************************************************************
*********************************************************************************
*******************************************************************\n")
           #print(f"\n\nFor Iteration No. {I}:\n\nMembers of Open-List are :
\n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

         print(f"\n\nSUCCESS MESSAGE!!!\n\nSTART STATE: {matrix}\n\nGOAL STATE :
{GS}\n\nWITH A* ALGORITHM, GOAL STATE IS REACHED IN ITERATION No. :'{I}' AND IT
APPEARS AFTER '{steps}' STEPS FROM INITIAL STATE, \n")

        Num_Explored_Nodes= len(clist)
           print(f"\nTotal Number of nodes explored to reach goal state :
{Num_Explored_Nodes}\n")

        #print(f"\n{CP_list}\n")

        def index_finder(element,lis) :
          i_element=0
          for x in range(len(lis)-1,0,-1) :
            if (CP_list[x][0]==element) :
              i_element=x
              break
          return i_element

        t=index_finder(GS,CP_list)
        while t:
          q1=CP_list[t][1]
          parent.append(q1)
          t=index_finder(q1,CP_list)
        parent.insert(0,GS)

        path=parent[::-1]

        print(f"\nTotal Number of States in the Optimal path : {len(path)}\n")
         print("\nIn A* Algorithm using No.s of misplaced tiles as a heuristic,
the path to goal state looks like this :\n")

        for y in range(0, len(path)-1,1) :
          print(f"\n{path[y]}==>\n")
        print(f"\n{GS}\n")

        heuristics_monotonicity_checker(path,2)



#print("\n***********************************************************************
*********************************************************************************
*********************************************************************")

#print("*************************************************************************
```

```
**********************************************************************************
*******************************************************************\n")

        return steps, I, path, Num_Explored_Nodes, olist, clist
        break

    neighbours=Move_Generator(popped)
     #print(f"\nFor Iteration {I}, and Popped State : {popped}, set of possible
neighbours : {neighbours}\n")

    olist_elements=[]
    clist_elements=[]

    for k in range (0, len(olist), 1) :
      olist_elements.append(olist[k][0])

    for k1 in range (0, len(clist), 1) :
      clist_elements.append(clist[k1][0])

    for neighbour in neighbours:
             if ((neighbour not in olist_elements) and (neighbour not in
clist_elements)) :
            olist.append((neighbour, steps+1))
            CP_list.append((neighbour,popped))



#*********************************************************************************
*****************************************
#*********************************************************************************
*****************************************

def astar_h3(matrix) :

  conv=convergence_calculator(matrix)

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]
  root=[a,b,c]

  clist=[]
  olist=[(root,0)]
  CP_list=[(root,"START")]
  I=0
  parent=[]


  while olist :

    I=I+1



#print("\n\n************************************************************************
```

```
************************************************************************
****************************************************************")

#print("****************************************************************
************************************************************************
************************************************************\n")

    print(f"\nIteration Number :{I}\n")


#print("\n**************************************************************
************************************************************************
**************************************************************")

#print("****************************************************************
************************************************************************
************************************************************\n")

       #print(f"\n\nFor Iteration No. {I}:\n\nMembers of Open-List are :
\n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

    heur=[]
    list_int=([],0)
    list_decider=[]

    for i in range (0, len(olist), 1) :
      pot_elmt, depth_pot_elmt = olist[i]
      score = h3(pot_elmt)
      decider = score + depth_pot_elmt
      list_int = (pot_elmt,decider)
      heur.append(list_int)

     #print(f"\nIn Iteration Number {I}, Members(n) of the Open-List and their
corresponding [g(n) + h2(n)] scores : {heur}\n")

    for i1 in range(0, len(heur), 1) :
      pot_decider = heur[i1][1]
      list_decider.append(pot_decider)

    min_decider=min(list_decider)
    #print(f"\nMinimum value of [g(n) + h2(n)] Score : {min_decider}\n")
    min_decider_ind=list_decider.index(min_decider)
     #print(f"\nAND it is appearing at index '{min_decider_ind}' of the Open
List\n")

    popped, steps = olist[min_decider_ind]
    clist.append(olist[min_decider_ind])
    olist.remove(olist[min_decider_ind])

    if (I==2000 and popped!=GS and conv==0) :
        print("\n\nFAILURE MESSAGE !!!\n\nSince we know beforehand from the
convergence-calculator function, that the given initial State will not converge,
we are exiting the A* Algorithm !!!\n\nThis is done due to hardware constraints
as the compilation platform is unable to keep computing until Open_List=[]
!!!\n\n")
```

```
      print("\n")
      return -1, I, -1, len(clist), olist, clist
      break

    if (popped==[[1,2,3],[4,5,6],[7,8,0]]) :

#print("\n*********************************************************************
**********************************************************************
*************************************************************")

#print("*********************************************************************
**********************************************************************
***********************************************************\n")
           #print(f"\n\nFor Iteration No. {I}:\n\nMembers of Open-List are :
\n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

         print(f"\n\nSUCCESS MESSAGE!!!\n\nSTART STATE: {matrix}\n\nGOAL STATE :
{GS}\n\nWITH A* ALGORITHM, GOAL STATE IS REACHED IN ITERATION No. :'{I}' AND IT
APPEARS AFTER '{steps}' STEPS FROM INITIAL STATE, \n")

        Num_Explored_Nodes= len(clist)
           print(f"\nTotal Number of nodes explored to reach goal state :
{Num_Explored_Nodes}\n")

        #print(f"\n{CP_list}\n")

        def index_finder(element,lis) :
          i_element=0
          for x in range(0,len(lis),1) :
            if (CP_list[x][0]==element) :
              i_element=x
              break
          return i_element

        t=index_finder(GS,CP_list)
        while t:
          q1=CP_list[t][1]
          parent.append(q1)
          t=index_finder(q1,CP_list)
        parent.insert(0,GS)

        path=parent[::-1]

        print(f"\nTotal Number of States in the Optimal path : {len(path)}\n")
        print("\nIn A* Algorithm using sum of manhattan distances as a heuristic,
the path to goal state looks like this :\n")

        for y in range(0, len(path)-1,1) :
          print(f"\n{path[y]}==>\n")
        print(f"\n{GS}\n")

        heuristics_monotonicity_checker(path,3)
```

```
#print("\n*****************************************************************************
********************************************************************************
*********************************************************************")

#print("******************************************************************************
********************************************************************************
*******************************************************************\n")

        return steps, I, path, Num_Explored_Nodes, olist, clist
        break

    neighbours=Move_Generator(popped)
     #print(f"\nFor Iteration {I}, and Popped State : {popped}, set of possible
neighbours : {neighbours}\n")

    olist_elements=[]
    clist_elements=[]

    for k in range (0, len(olist), 1) :
      olist_elements.append(olist[k][0])

    for k1 in range (0, len(clist), 1) :
      clist_elements.append(clist[k1][0])

    for neighbour in neighbours:
             if  ((neighbour  not  in  olist_elements)  and  (neighbour  not  in
clist_elements)) :
            olist.append((neighbour, steps+1))
            CP_list.append((neighbour,popped))



################################################################################
#########################################

def astar_h4(matrix) :

  conv=convergence_calculator(matrix)

  a=matrix[0][:]
  b=matrix[1][:]
  c=matrix[2][:]
  root=[a,b,c]

  clist=[]
  olist=[(root,0)]
  CP_list=[(root,"START")]
  I=0
  parent=[]


  while olist :

    I=I+1
```

```
#print("\n\n*********************************************************************
*********************************************************************************
***********************************************************************")

#print("*************************************************************************
*********************************************************************************
*******************************************************************\n")

    print(f"\nIteration Number :{I}\n")


#print("\n***********************************************************************
*********************************************************************************
*********************************************************************")

#print("*************************************************************************
*********************************************************************************
*******************************************************************\n")

       #print(f"\n\nFor Iteration No. {I}:\n\nMembers of Open-List are :
\n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

    heur=[]
    list_int=([],0)
    list_decider=[]

    for i in range (0, len(olist), 1) :
      pot_elmt, depth_pot_elmt = olist[i]
      score = h4(pot_elmt)
      decider = score + depth_pot_elmt
      list_int = (pot_elmt,decider)
      heur.append(list_int)

     #print(f"\nIn Iteration Number {I}, Members(n) of the Open-List and their
corresponding [g(n) + h2(n)] scores : {heur}\n")

    for i1 in range(0, len(heur), 1) :
      pot_decider = heur[i1][1]
      list_decider.append(pot_decider)

    min_decider=min(list_decider)
    #print(f"\nMinimum value of [g(n) + h2(n)] Score : {min_decider}\n")
    min_decider_ind=list_decider.index(min_decider)
     #print(f"\nAND it is appearing at index '{min_decider_ind}' of the Open
List\n")

    popped, steps = olist[min_decider_ind]
    clist.append(olist[min_decider_ind])
    olist.remove(olist[min_decider_ind])

    if (I==2000 and popped!=GS and conv==0) :
        print("\n\nFAILURE MESSAGE !!!\n\nSince we know beforehand from the
convergence-calculator function, that the given initial State will not converge,
```

```
we are exiting the A* Algorithm !!!\n\nThis is done due to hardware constraints as the compilation platform is unable to keep computing until Open_List=[] !!!\n\n")
      print("\n")
      return -1, I, -1, len(clist), olist, clist
      break


    if (popped==[[1,2,3],[4,5,6],[7,8,0]]) :

#print("\n*********************************************************************************************************************************************************************************************************")

#print("*****************************************************************************************************************************************************************************************************\n")
           #print(f"\n\nFor Iteration No. {I}:\n\nMembers of Open-List are : \n{olist}\n\nMembers of Closed-List are : \n{clist}\n\n")

         print(f"\n\nSUCCESS MESSAGE!!!\n\nSTART STATE: {matrix}\n\nGOAL STATE : {GS}\n\nWITH A* ALGORITHM, GOAL STATE IS REACHED IN ITERATION No. :'{I}' AND IT APPEARS AFTER '{steps}' STEPS FROM INITIAL STATE, \n")

        Num_Explored_Nodes= len(clist)
           print(f"\nTotal Number of nodes explored to reach goal state : {Num_Explored_Nodes}\n")

        #print(f"\n{CP_list}\n")

        def index_finder(element,lis) :
          i_element=0
          for x in range(len(lis)-1,0,-1) :
            if (CP_list[x][0]==element) :
              i_element=x
              break
          return i_element

        t=index_finder(GS,CP_list)
        while t:
          q1=CP_list[t][1]
          parent.append(q1)
          t=index_finder(q1,CP_list)
        parent.insert(0,GS)

        path=parent[::-1]

        print(f"\nTotal Number of States in the Optimal path : {len(path)}\n")
         print("\nIn A* Algorithm using relative position squared weightage, as a heuristic, the path to goal state looks like this :\n")

        for y in range(0, len(path)-1,1) :
          print(f"\n{path[y]}==>\n")
        print(f"\n{GS}\n")
```

```
        heuristics_monotonicity_checker(path,4)



#print("\n***********************************************************************
*********************************************************************************
*********************************************************************")

#print("*************************************************************************
*********************************************************************************
*******************************************************************\n")

        return steps, I, path, Num_Explored_Nodes, olist, clist
        break

    neighbours=Move_Generator(popped)
     #print(f"\nFor Iteration {I}, and Popped State : {popped}, set of possible
neighbours : {neighbours}\n")

    olist_elements=[]
    clist_elements=[]

    for k in range (0, len(olist), 1) :
      olist_elements.append(olist[k][0])

    for k1 in range (0, len(clist), 1) :
      clist_elements.append(clist[k1][0])

    for neighbour in neighbours:
             if  ((neighbour  not  in  olist_elements)  and  (neighbour  not  in
clist_elements)) :
            olist.append((neighbour, steps+1))
            CP_list.append((neighbour,popped))



#################################################################################
#########################################

def  explored_states_inclusivity_checker  (LIST_1,LIST_2):  #LIST_1  is  closed  list
of superior heuristic, LIST_2 is closed list of inferior heuristic
  clist_elements_1 = []
  clist_elements_2 = []

  for k1 in range(0, len(LIST_1), 1):
    clist_elements_1.append(LIST_1[k1][0])
  for k2 in range(0, len(LIST_2), 1):
    clist_elements_2.append(LIST_2[k2][0])

  inclusivity_counter=0

  for element in clist_elements_1 :
    if element in clist_elements_2:
      inclusivity_counter=inclusivity_counter+1
    #else :
```

```
      #print(element)
      #indx=clist_elements_1.index(element)
      #print(indx)

  #print (len(clist_elements_1))

  #print (len(clist_elements_2))

  #print (inclusivity_counter)

  if inclusivity_counter==len(LIST_1) :
     print("\n\nAll states explored by Superior Heuristics are also expanded by
Inferior heuristics\n\n")
  else :
    print ("\n\nInclusivity Check Failed!!!")

##################################################################################
#########################################

tracemalloc.start()
BEGINNING=time.time()

##################################################################################
#########################################

IS2=[[1,2,3],[4,5,6],[0,7,8]]
#h2(n): Converging, Iterations : 3, Steps : 2, Time : 0.0017s, Peak RAM : 24.68kB

IS3=[[1,2,3],[4,0,5],[7,8,6]]
#h2(n): Converging, Iterations : 3, Steps : 2, Time : 0.0019s, Peak RAM : 25.40kB

IS5=[[1,2,0],[4,5,3],[7,8,6]]
#h2(n): Converging, Iterations : 3, Steps : 2, Time : 0.0015s, Peak RAM : 25.20kB

##################################################################################
#########################################

IS1 = [[0,1,2],[4,5,3],[7,8,6]]
#h1(n): Converging, Iterations : 30, Steps : 4, Time : 0.033s, Peak RAM : 54.34kB
#h2(n): Converging, Iterations : 5, Steps : 4, Time : 0.026s, Peak RAM : 38.36kB
#h3(n): Converging, Iterations : 5, Steps : 4, Time : 0.012s, Peak RAM : 30.55kB
#h4(n): Converging, Iterations :5, Steps :4, Time : 0.018s, Peak RAM : 33.95kB

##################################################################################
#########################################

IS4=[[4,1,2],[7,5,3],[8,0,6]]
#h1(n): Converging, Iterations : 140, Steps : 7, Time : 0.08s, Peak RAM :
2654.9kB
#h2(n): Converging, Iterations : 8, Steps : 7, Time : 0.06s, Peak RAM : 40.98kB
#h3(n): Converging, Iterations : 8, Steps : 7, Time : 0.02s, Peak RAM : 34.34kB
#h4(n): Converging, Iterations : 9, Steps :7, Time : 0.036s, Peak RAM : 37.07kB

##################################################################################
#########################################
```

```
IS6=[[5,0,8],[4,2,1],[7,3,6]]
#h1(n): Converging, Iterations : 64353, Steps : 21, Time : 11609.743s, Peak RAM : 42834.66kB
#h2(n): Converging, Iterations : 5782, Steps : 21, Time : 145.91s, Peak RAM : 4304.39kB
#h3(n): Converging, Iterations : 2094, Steps : 21, Time : 172.41s, Peak RAM : 1598.38kB
#h4(n): Converging, Iterations : 1075, Steps : 27, Time : 13.98, Peak RAM : 814.02kB

##########################################################################################################################

IS7=[[0,1,2],[3,4,5],[6,7,8]]
#h2(n): Converging, Iterations : 8300, Steps : 22, Time : 369.89s, Peak RAM : 6231.75kB
#h3(n): Converging, Iterations : 1306, Steps : 22, Time : 70.06s, Peak RAM : 2965.59kB
#h4(n): Converging, Iterations : 2275, Steps : 22, Time : 63.53s, Peak RAM : 1722.46kB

##########################################################################################################################

ISN=[[1,2,3],[4,5,6],[8,7,0]]
#h1(n): Non-Converging
#h2(n): Non-Converging
#h3(n): Non-Converging
#h4(n): Non-Converging


##########################################################################################################################

ISQ=[[3,2,1],[4,5,6],[8,7,0]]
#h1(n): Converging, Iterations : 130544, Steps : 24, Time : 19988.6s, Peak RAM : 185036.82kB
#h2(n): Converging, Iterations : 18700, Steps : 24, Time : 1115.48s, Peak RAM : 11679.06kB
#h3(n): Converging, Iterations : 3299, Steps : 24, Time : 415.66s, Peak RAM : 2617.29kB
#h4(n): Converging, Iterations : 1824, Steps : 38, Time : 38.79s, Peak RAM : 1387.54kB

[STEPS_1, ITERATIONS_1, RECONSTRUCTED_PATH_1, No_of_EXPLORED_NODES_1, OPEN_LIST_1, CLOSED_LIST_1] = astar_h1(ISQ)
[STEPS_2, ITERATIONS_2, RECONSTRUCTED_PATH_2, No_of_EXPLORED_NODES_2, OPEN_LIST_2, CLOSED_LIST_2] = astar_h2(ISQ)
[STEPS_3, ITERATIONS_3, RECONSTRUCTED_PATH_3, No_of_EXPLORED_NODES_3, OPEN_LIST_3, CLOSED_LIST_3] = astar_h3(ISQ)
[STEPS_4, ITERATIONS_4, RECONSTRUCTED_PATH_4, No_of_EXPLORED_NODES_4, OPEN_LIST_4, CLOSED_LIST_4] = astar_h4(ISQ)
```

```
#######################################################################################
##########################################

#Demo for verifying that all the states expanded by better heuristics are also developed by inferior heuristics.

[STEPS_2, ITERATIONS_2, RECONSTRUCTED_PATH_2, No_of_EXPLORED_NODES_2, OPEN_LIST_2, CLOSED_LIST_2] = astar_h2(IS7)
[STEPS_3, ITERATIONS_3, RECONSTRUCTED_PATH_3, No_of_EXPLORED_NODES_3, OPEN_LIST_3, CLOSED_LIST_3] = astar_h3(IS7)

explored_states_inclusivity_checker(CLOSED_LIST_3, CLOSED_LIST_2)

#**************************************************************************************
******************************************
#**************************************************************************************
******************************************

CONCLUSION=time.time()

TOTAL_TIME=CONCLUSION-BEGINNING

print(f"\nTotal Time taken in this execution is :{TOTAL_TIME} seconds")

CURRENT_MEM, PEAK_MEM = tracemalloc.get_traced_memory()

print(f"\nRAM CONSUMPTION DETAILS :\nCurrent Memory Consumption is : {CURRENT_MEM/1024} kB and the Peak Memory Consumption is : {PEAK_MEM/1024} kB")

tracemalloc.stop()
```

Chapter 2:
Post-Run Analytics and Assignment Questions

Q.: Tasks :

1. **Observe and verify that better heuristics expand lesser states.**
2. **Observe and verify that all the states expanded by better heuristics should also be developed by inferior heuristics.**
3. **Observe un-reachability and provide proof.**
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.

5. **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.**

Answers :

1. We know that h3>h2>h1 in terms of heuristics performance in the 8-Puzzle problem. The Observations for no. of iterations (which is also equal to the number of explored states) for all 3 heuristics corresponding to Initial State ISQ is as follows:

```
ISQ=[[3,2,1],[4,5,6],[8,7,0]]

#h1(n): Converging, Iterations : 130544, Steps : 24, Time : 19988.6s, Peak RAM : 185036.82kB

#h2(n): Converging, Iterations : 18700, Steps : 24, Time : 1115.48s, Peak RAM : 11679.06kB

#h3(n): Converging, Iterations : 3299, Steps : 24, Time : 415.66s, Peak RAM : 2617.29kB

#h4(n): Converging, Iterations : 1824, Steps : 38, Time : 38.79s, Peak RAM : 1387.54kB
```

2. The function implemented in code : **explored_states_inclusivity_checker (LIST_1, LIST_2)** verifies that all states expanded by superior heuristics are included in the states expanded by inferior heuristics.
Example : For Initial State IS7 :

```
IS7=[[0,1,2],[3,4,5],[6,7,8]]

#h2(n): Converging, Iterations : 8300, Steps : 22, Time : 369.89s, Peak RAM : 6231.75kB

#h3(n): Converging, Iterations : 1306, Steps : 22, Time : 70.06s, Peak RAM : 2965.59kB

#h4(n): Converging, Iterations : 2275, Steps : 22, Time : 63.53s, Peak RAM : 1722.46kB
```

```
explored_states_inclusivity_checker(CLOSED_LIST_3, CLOSED_LIST_2)
```

```
1306
8300
1306


All states explored by Superior Heuristics are also expanded by Inferior heuristics


0
```

3. Unreachability & Proof :

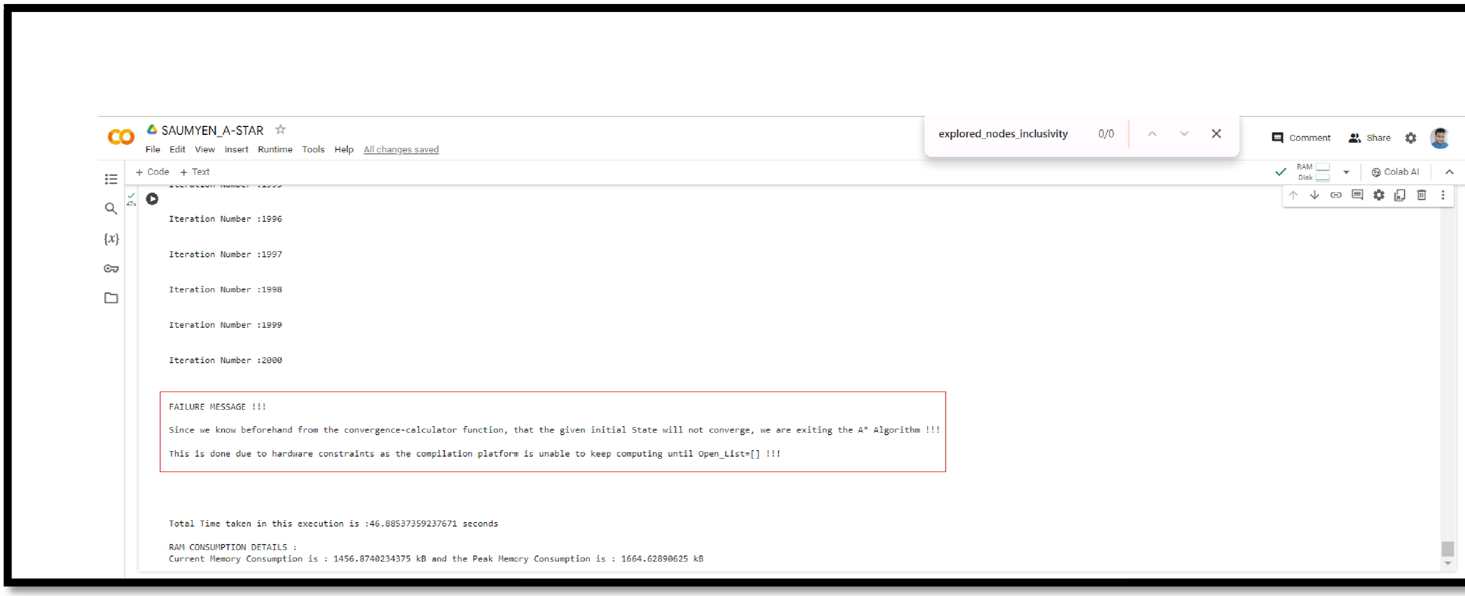
For Initial State ISN:

```
ISN=[[1,2,3],[4,5,6],[8,7,0]]

#h1(n): Non-Converging
#h2(n): Non-Converging
#h3(n): Non-Converging
#h4(n): Non-Converging
```

```
convergence_calculator (ISN)
```

```
Given Matrix : [[1, 2, 3], [4, 5, 6], [8, 7, 0]] has 1 numbers of inversions and since it's an ODD Integer, this 8-Puzzle Problem will NOT CONVERGE to the Goal State!
0
```



4. Monotonicity Verification: (Using IS7 Initial State)

Output for h2:

Iteration Number :8298

Iteration Number :8299

Iteration Number :8300

SUCCESS MESSAGE!!!

START STATE: [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

GOAL STATE : [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

WITH A* ALGORITHM, GOAL STATE IS REACHED IN ITERATION No. :'8300' AND IT APPEARS AFTER '22' STEPS FROM INITIAL STATE,

Total Number of nodes explored to reach goal state : 8300

Total Number of States in the Optimal path : 23

In A* Algorithm using No.s of misplaced tiles as a heuristic, the path to goal state looks like this :

[[0, 1, 2], [3, 4, 5], [6, 7, 8]]==>

[[1, 0, 2], [3, 4, 5], [6, 7, 8]]==>

[[1, 4, 2], [3, 0, 5], [6, 7, 8]]==>

[[1, 4, 2], [0, 3, 5], [6, 7, 8]]==>

[[1, 4, 2], [6, 3, 5], [0, 7, 8]]==>

[[1, 4, 2], [6, 3, 5], [7, 0, 8]]==>

[[1, 4, 2], [6, 3, 5], [7, 8, 0]]==>

[[1, 4, 2], [6, 3, 0], [7, 8, 5]]==>

[[1, 4, 2], [6, 0, 3], [7, 8, 5]]==>

[[1, 4, 2], [0, 6, 3], [7, 8, 5]]==>

[[1, 4, 2], [7, 6, 3], [0, 8, 5]]==>

[[1, 4, 2], [7, 6, 3], [8, 0, 5]]==>

[[1, 4, 2], [7, 0, 3], [8, 6, 5]]==>

[[1, 0, 2], [7, 4, 3], [8, 6, 5]]==>

[[1, 2, 0], [7, 4, 3], [8, 6, 5]]==>

[[1, 2, 3], [7, 4, 0], [8, 6, 5]]==>

[[1, 2, 3], [7, 4, 5], [8, 6, 0]]==>

[[1, 2, 3], [7, 4, 5], [8, 0, 6]]==>

[[1, 2, 3], [7, 4, 5], [0, 8, 6]]==>

[[1, 2, 3], [0, 4, 5], [7, 8, 6]]==>

[[1, 2, 3], [4, 0, 5], [7, 8, 6]]==>

[[1, 2, 3], [4, 5, 0], [7, 8, 6]]==>

[[1, 2, 3], [4, 5, 6], [7, 8, 0]]

h2(n) or Numbers of Misplaced Tiles is a consistent & monotonic Heuristic!

Total Time taken in this execution is :280.6962699890137 seconds

RAM CONSUMPTION DETAILS :
Current Memory Consumption is : 5072.625 kB and the Peak Memory Consumption is : 6169.4013671875 kB

<u>Output for h3:</u>

```
Iteration Number :1300

Iteration Number :1301

Iteration Number :1302

Iteration Number :1303

Iteration Number :1304

Iteration Number :1305

Iteration Number :1306


SUCCESS MESSAGE!!!

START STATE: [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

GOAL STATE : [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

WITH A* ALGORITHM, GOAL STATE IS REACHED IN ITERATION No. :'1306' AND IT APPEARS AFTER '22' STEPS FROM INITIAL STATE,

Total Number of nodes explored to reach goal state : 1306

Total Number of States in the Optimal path : 23

In A* Algorithm using sum of manhattan distances as a heuristic, the path to goal state looks like this :

[[0, 1, 2], [3, 4, 5], [6, 7, 8]]==>

[[1, 0, 2], [3, 4, 5], [6, 7, 8]]==>

[[1, 4, 2], [3, 0, 5], [6, 7, 8]]==>

[[1, 4, 2], [0, 3, 5], [6, 7, 8]]==>

[[1, 4, 2], [6, 3, 5], [0, 7, 8]]==>

[[1, 4, 2], [6, 3, 5], [7, 0, 8]]==>

[[1, 4, 2], [6, 3, 5], [7, 8, 0]]==>

[[1, 4, 2], [6, 3, 0], [7, 8, 5]]==>

[[1, 4, 2], [6, 0, 3], [7, 8, 5]]==>

[[1, 4, 2], [0, 6, 3], [7, 8, 5]]==>

[[1, 4, 2], [7, 6, 3], [0, 8, 5]]==>

[[1, 4, 2], [7, 6, 3], [8, 0, 5]]==>

[[1, 4, 2], [7, 0, 3], [8, 6, 5]]==>

[[1, 0, 2], [7, 4, 3], [8, 6, 5]]==>

[[1, 2, 0], [7, 4, 3], [8, 6, 5]]==>

[[1, 2, 3], [7, 4, 0], [8, 6, 5]]==>

[[1, 2, 3], [7, 4, 5], [8, 6, 0]]==>

[[1, 2, 3], [7, 4, 5], [8, 0, 6]]==>

[[1, 2, 3], [7, 4, 5], [0, 8, 6]]==>

```
[[1, 2, 3], [0, 4, 5], [7, 8, 6]]==>

[[1, 2, 3], [4, 0, 5], [7, 8, 6]]==>

[[1, 2, 3], [4, 5, 0], [7, 8, 6]]==>

[[1, 2, 3], [4, 5, 6], [7, 8, 0]]

h3(n) or Sum of Manhattan distances is a consistent & monotonic Heuristic!

Total Time taken in this execution is :64.19942808151245 seconds

RAM CONSUMPTION DETAILS :
        Current Memory Consumption is : 922.1083984375 kB and the Peak Memory Consumption is :
        998.4013671875 kB
```

5. If we add the cost of the blank tile, the heuristics both h2 & h3 are becoming inconsistent and losing the monotone restriction property. Since we are already calculating the cost of the blank tile movement in the backward cost, adding its relative position w.r.t. Goal State will lead to double additions of similar costs.

   Also, since the positions of all the different misplaced tiles are solved only by movement of the blank tile, we welcome such movements even if it means that the blank tile is moving farther and farther away from the bottom right corner of the matrix. So, if we choose to include blank tile cost in either of the heuristics h2 or h3, we will deliberately reject exploration of nodes which might result in moving the blank tile away from the bottom right corner and solving the positions of the other non-zero tiles.

Q.: 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.

a.) For IS1 = [[0,1,2],[4,5,3],[7,8,6]]

	A* Using h1(n)	A* Using h2(n)	A* Using h3(n)	A* Using h4(n)
No. of Iterations to Convergence	30	5	5	5
No. of Explored nodes	30	5	5	5
Steps from Root Node	4	4	4	4
Time Taken (in s)	0.03	0.02	0.012	0.018
Peak RAM Consumption (in kB)	54	38	30	34

b.) For IS4 = [[4,1,2],[7,5,3],[8,0,6]]

	A* Using h1(n)	A* Using h2(n)	A* Using h3(n)	A* Using h4(n)
No. of Iterations to Convergence	140	8	8	9
No. of Explored nodes	140	8	8	9
Steps from Root Node	7	7	7	7
Time Taken (in s)	0.08	0.06	0.02	0.036
Peak RAM Consumption (in kB)	2654	40	34	37

c.) For IS6 = [[4,1,2],[7,5,3],[8,0,6]]

	A* Using h1(n)	A* Using h2(n)	A* Using h3(n)	A* Using h4(n)
No. of Iterations to Convergence	64353	5782	2094	1075
No. of Explored nodes	64353	5782	2094	1075
Steps from Root Node	21	21	21	27
Time Taken (in s)	11609.7	146	172	14
Peak RAM Consumption (in kB)	42834	4304	1598	814

d.) For IS7 = [[0,1,2],[3,4,5],[6,7,8]]

	A* Using h2(n)	A* Using h3(n)	A* Using h4(n)
No. of Iterations to Convergence	8300	1306	2275
No. of Explored nodes	8300	1306	2275
Steps from Root Node	22	22	22
Time Taken (in s)	370	70	63
Peak RAM Consumption (in kB)	6231	2965	1722

e.) For ISQ = [[3,2,1],[4,5,6],[8,7,0]]

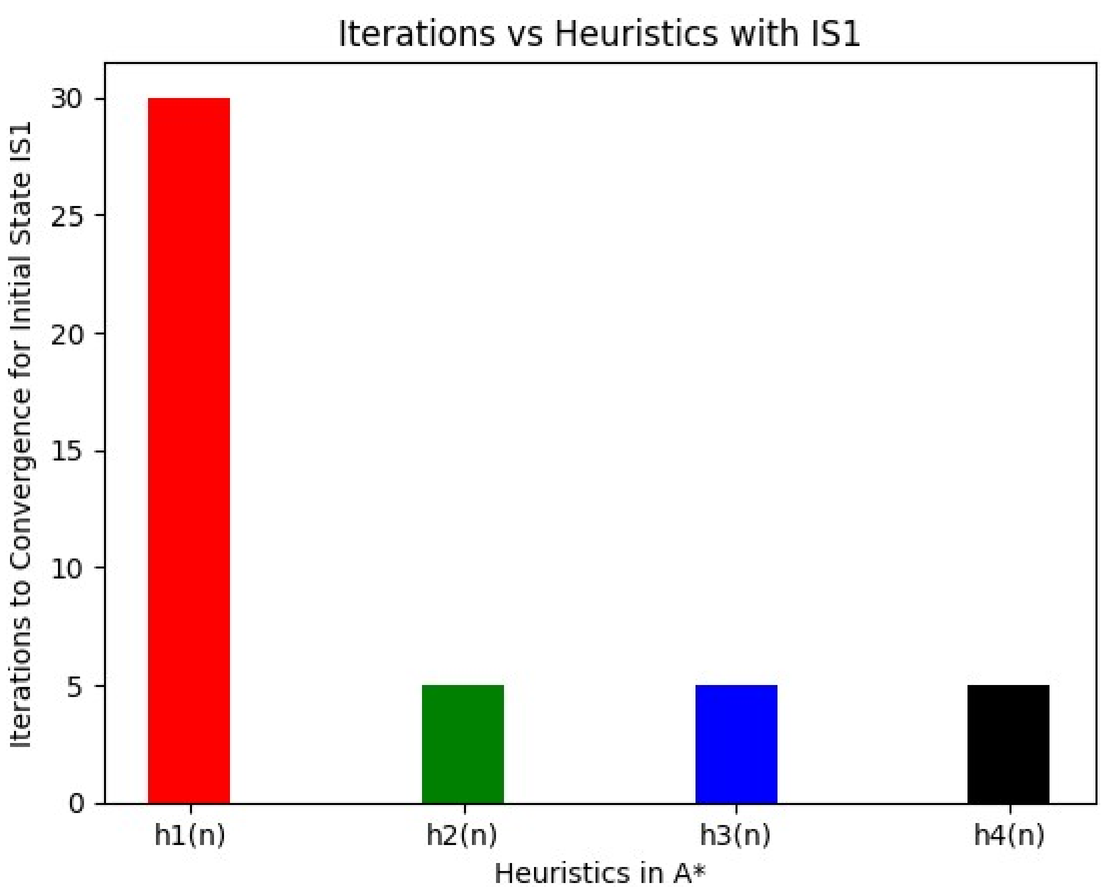
	A* Using h1(n)	A* Using h2(n)	A* Using h3(n)	A* Using h4(n)
No. of Iterations to Convergence	130544	18700	3299	1824
No. of Explored nodes	130544	18700	3299	1824
Steps from Root Node	24	24	24	38
Time Taken (in s)	19988	1115	415	38
Peak RAM Consumption (in kB)	185036	11679	2617	1387

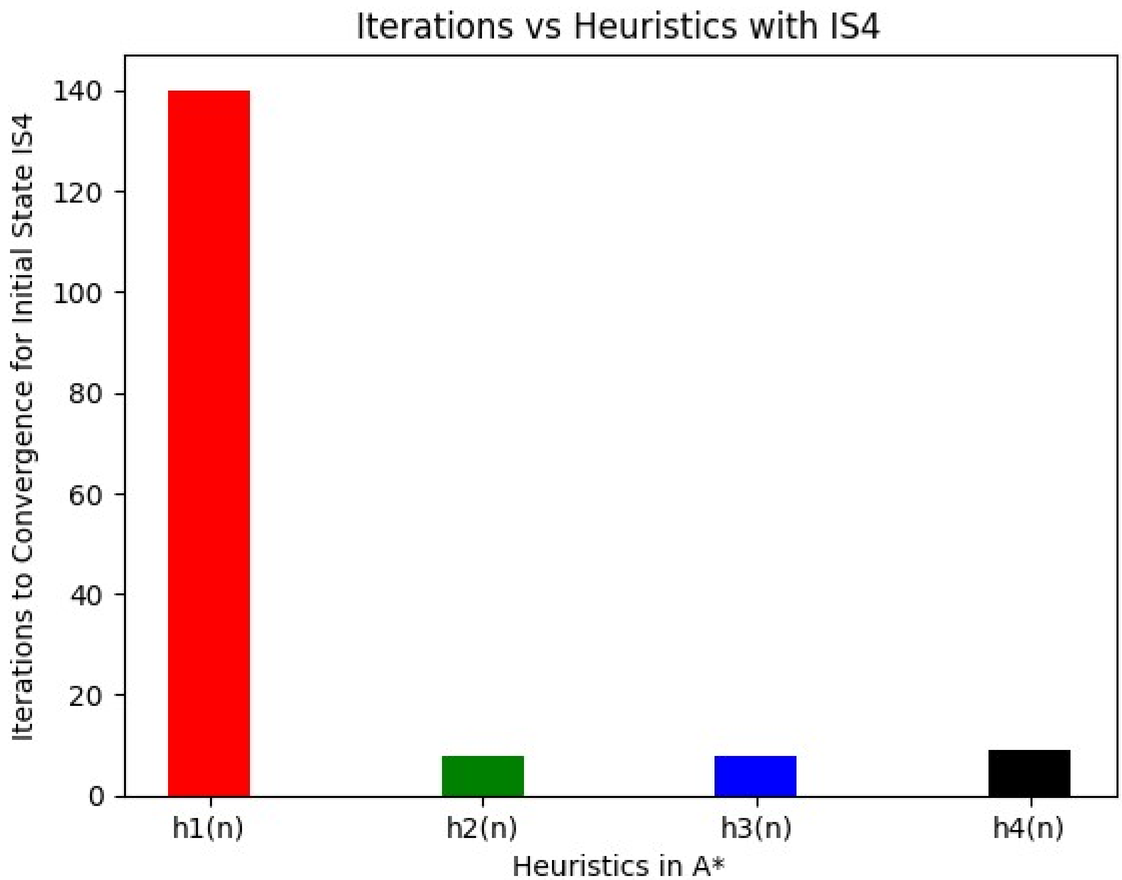
Initial States Chosen:

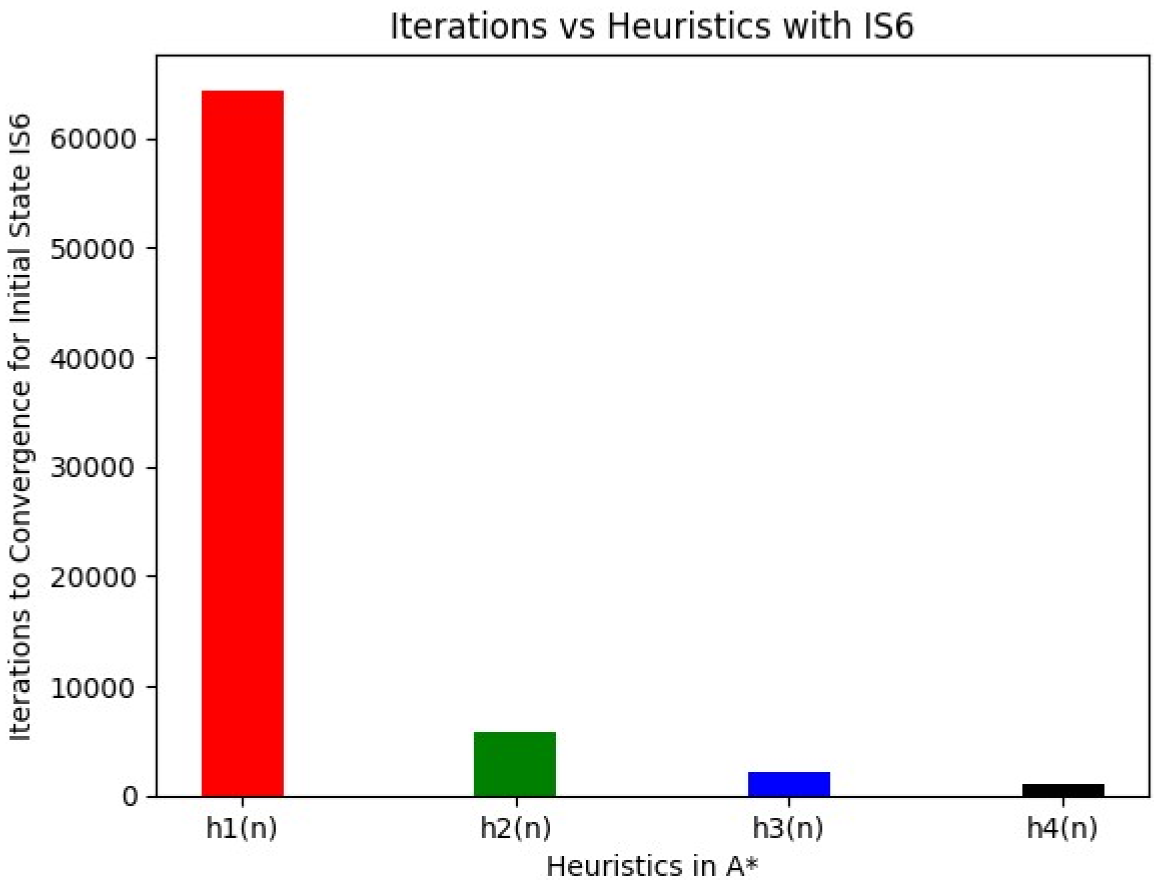
IS1 = [[0,1,2],[4,5,3],[7,8,6]]
IS4 = [[4,1,2],[7,5,3],[8,0,6]]
IS6 = [[4,1,2],[7,5,3],[8,0,6]]
IS7 = [[0,1,2],[3,4,5],[6,7,8]]
ISQ = [[3,2,1],[4,5,6],[8,7,0]]

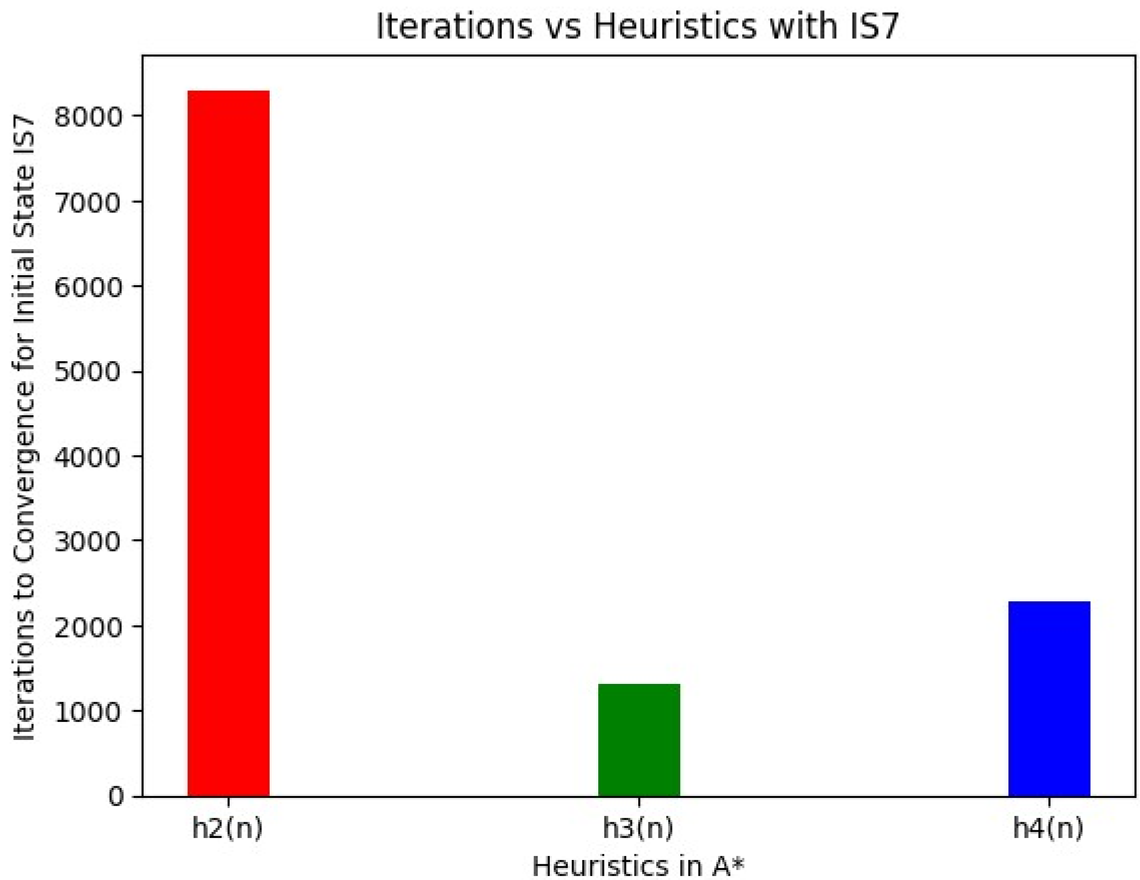
Graphical Analysis:

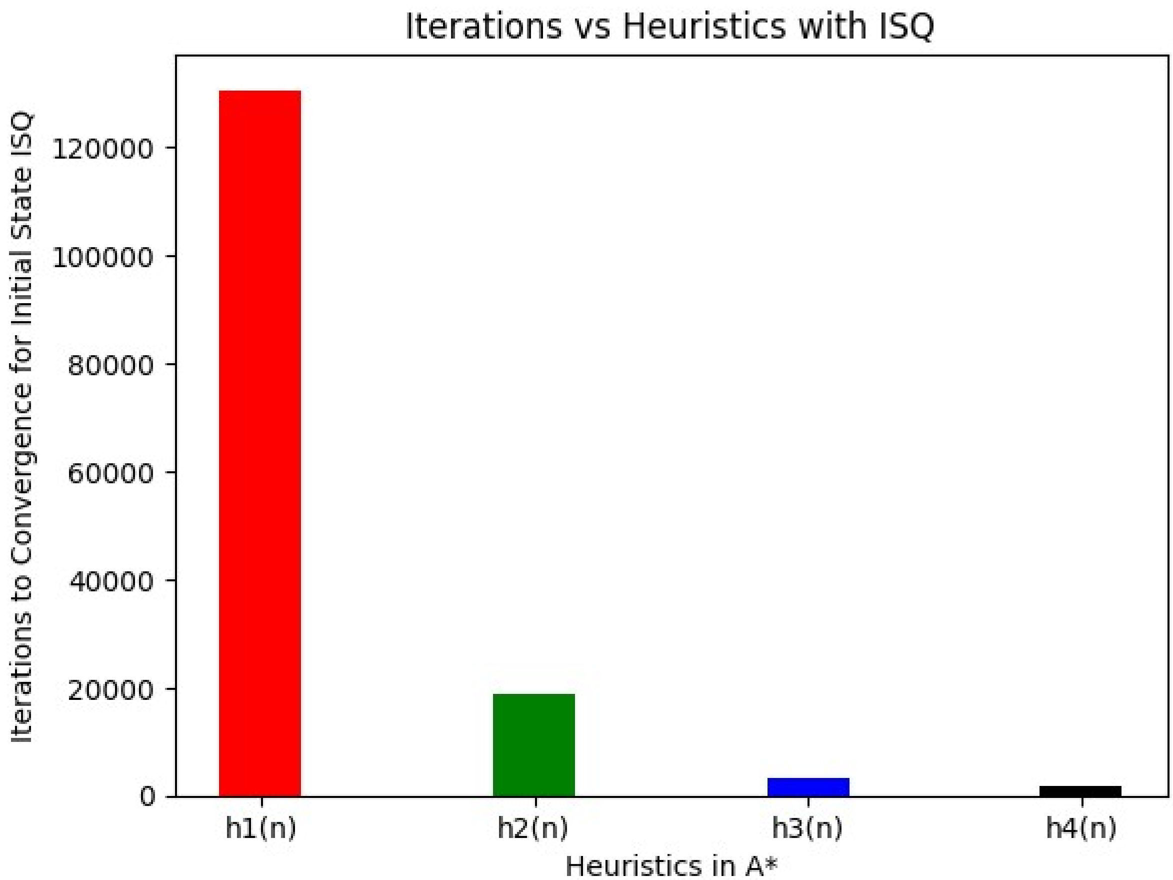
a.) Iterations or Number of Nodes Explored



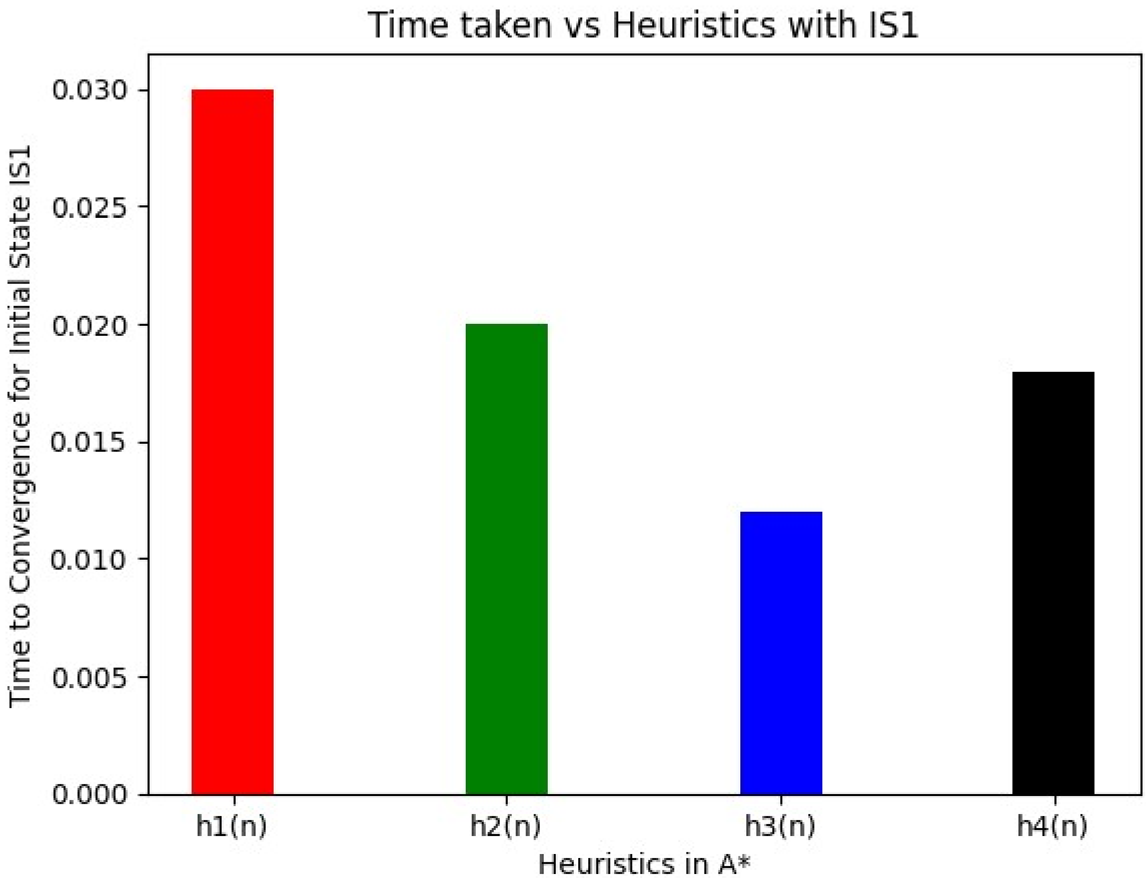

Iterations vs Heuristics with IS4
Iterations to Convergence for Initial State IS4
140
120
100
80
60
40
20
0
h1(n)
h2(n)
h3(n)
h4(n)
Heuristics in A*

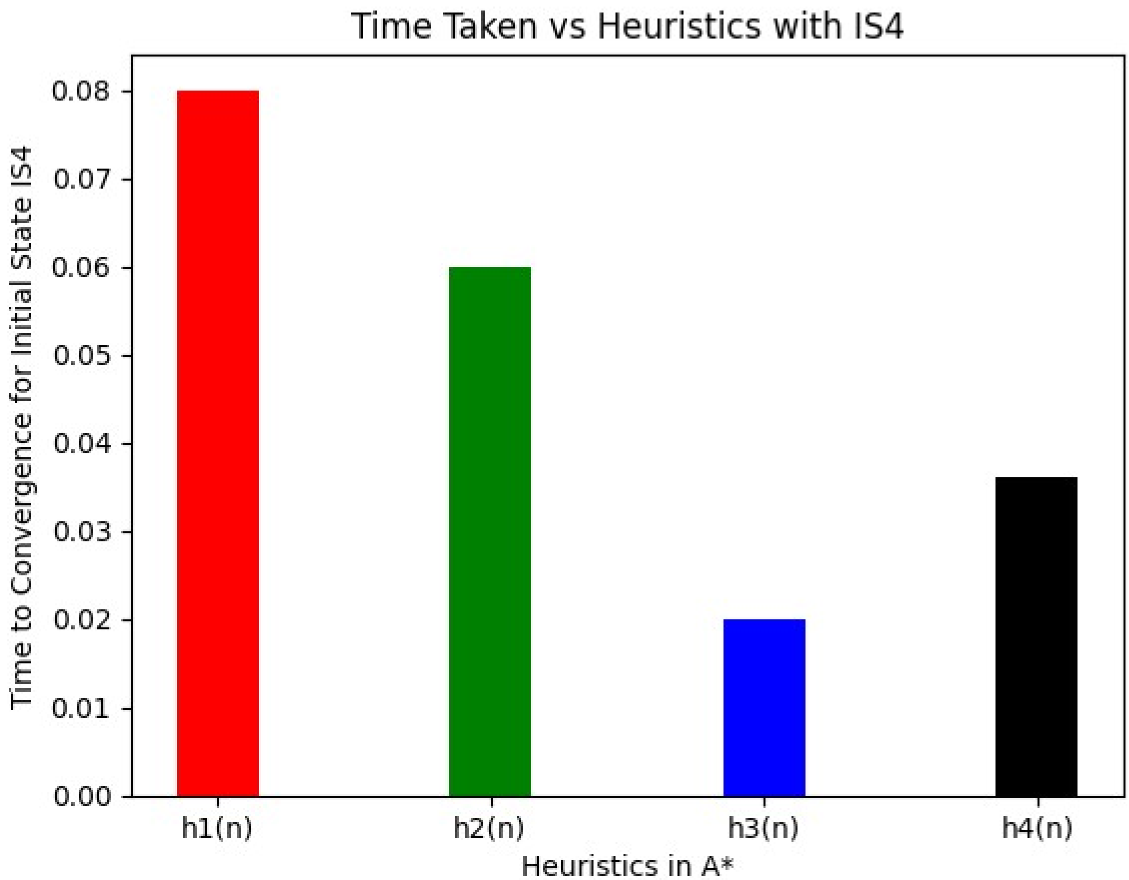

Iterations vs Heuristics with IS6
Iterations to Convergence for Initial State IS6
60000
50000
40000
30000
20000
10000
0
h1(n)
h2(n)
h3(n)
h4(n)
Heuristics in A*

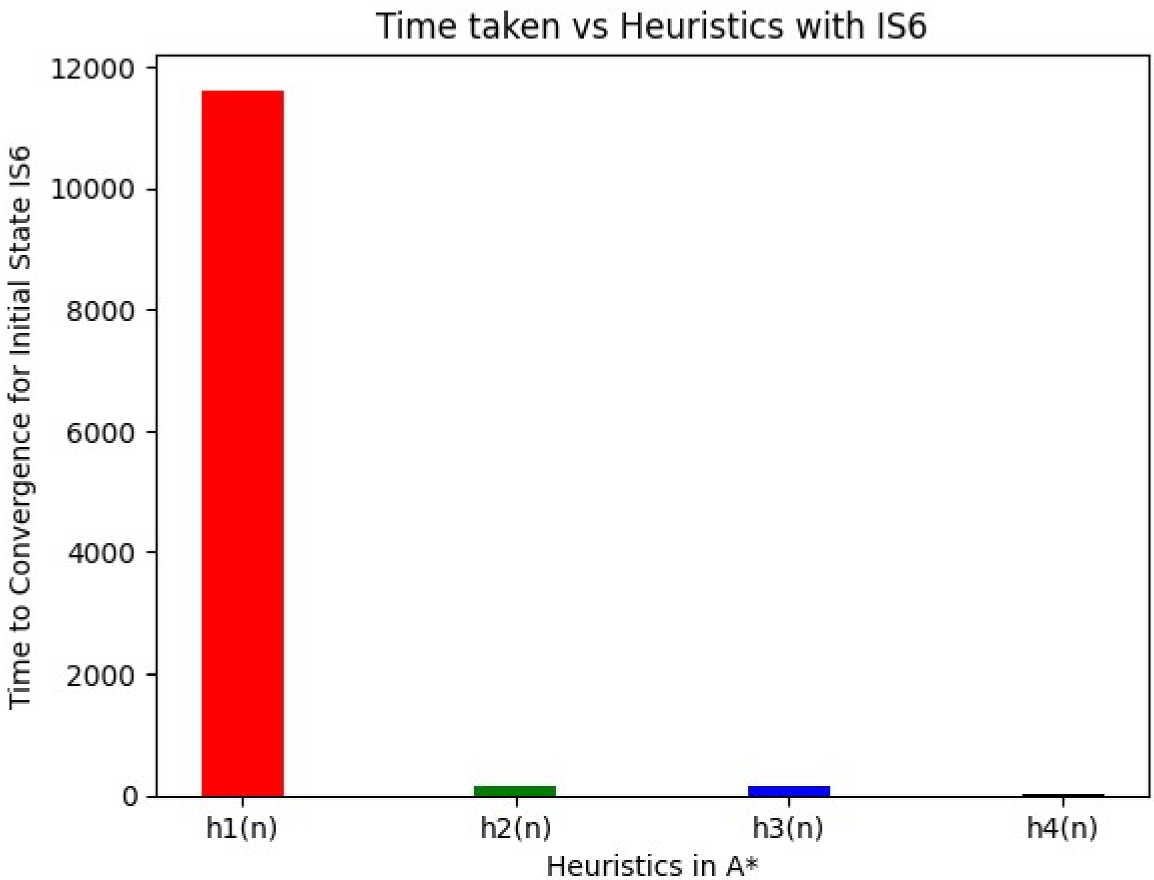

Iterations vs Heuristics with IS7
Iterations to Convergence for Initial State IS7
8000
7000
6000
5000
4000
3000
2000
1000
0
h2(n)
h3(n)
h4(n)
Heuristics in A*

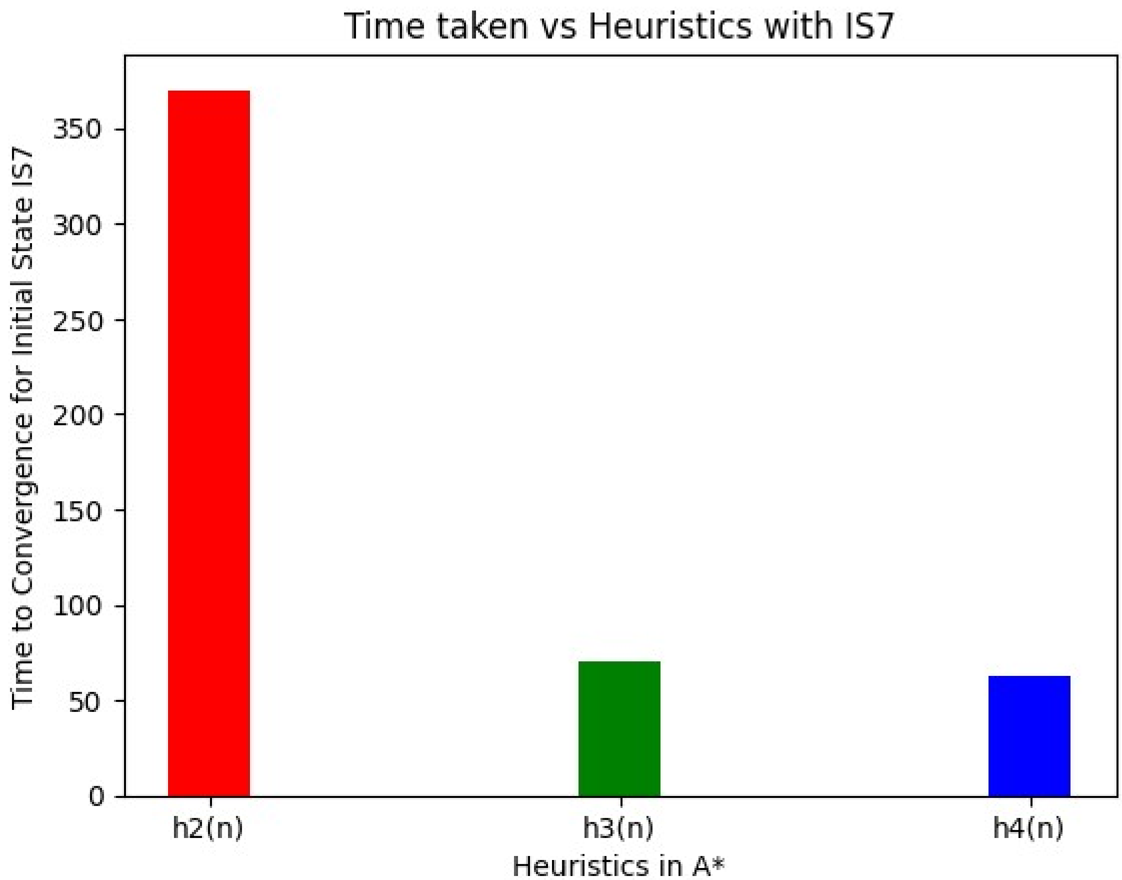

Iterations vs Heuristics with ISQ
Iterations to Convergence for Initial State ISQ
120000
100000
80000
60000
40000
20000
0
h1(n)
h2(n)
h3(n)
h4(n)
Heuristics in A*

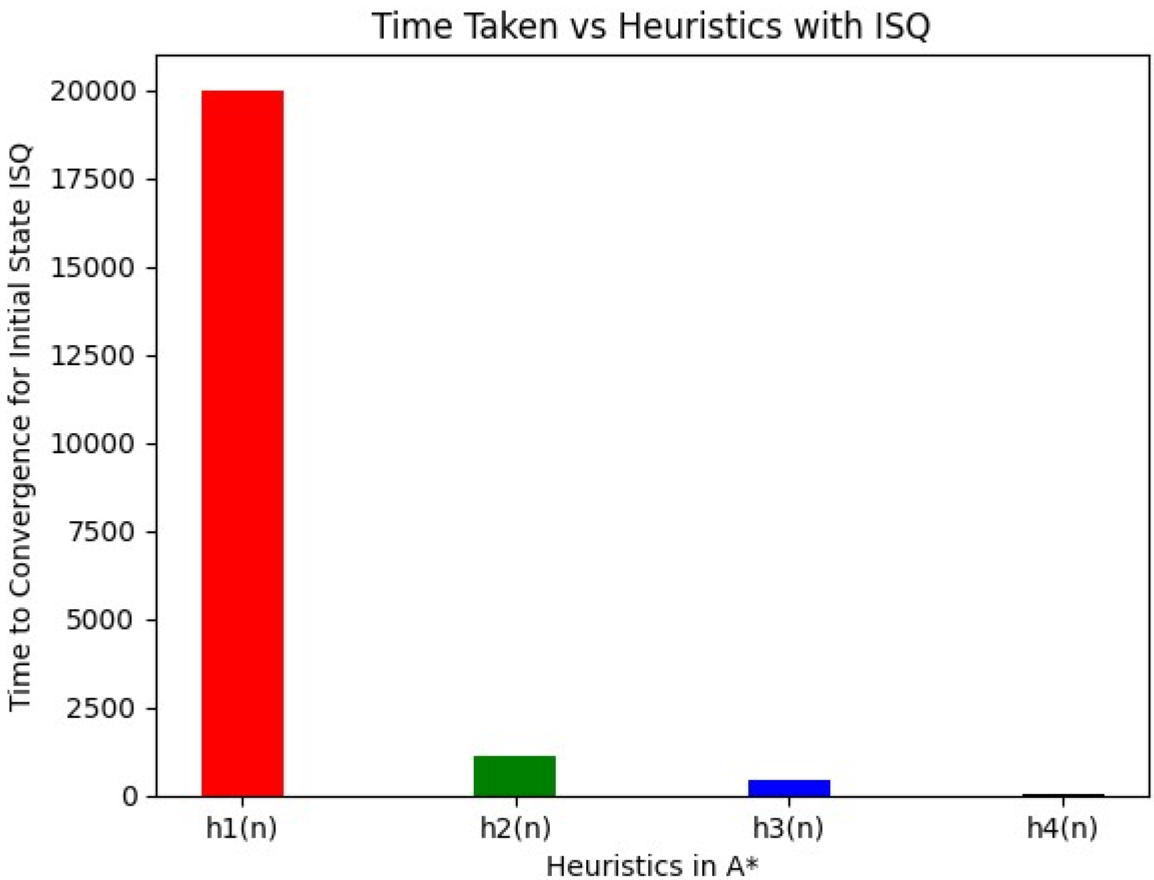
b.) **Time Taken to Converge**




Time Taken vs Heuristics with IS4
Time to Convergence for Initial State IS4
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
h1(n)
h2(n)
h3(n)
h4(n)
Heuristics in A*


Time taken vs Heuristics with IS6
Time to Convergence for Initial State IS6
12000
10000
8000
6000
4000
2000
0
h1(n)
h2(n)
h3(n)
h4(n)
Heuristics in A*


Time taken vs Heuristics with IS7
Time to Convergence for Initial State IS7
350
300
250
200
150
100
50
0
h2(n)
h3(n)
h4(n)
Heuristics in A*



Comments Regarding Optimality & Time Taken:

1. A* Search Algorithm which selects an intermediary node between Start & Goal State based on the sum of Backward and Forward Costs, expands the least number of states to find the optimal path.
2. Optimality is guaranteed in A* Search if the heuristics chosen for the estimation is 'optimistic', 'consistent', 'monotonically non-increasing' & 'admissible'.
3. Unlike Informed Search Algorithms & Local Search Algorithms like the Greedy Best First Search, Hill-Climbing etc., A* Search delivers optimal solutions every-time as it considers both backward and forward costs.
4. Performance of A* Search w.r.t No. of nodes explored, Time taken and RAM Consumed, largely depends on the quality of heuristics chosen to optimistically estimate the forward cost.
5. In case of the 8-Puzzle Problem, h2(n) or no. of misplaced tiles & h3(n) Sum of Manhattan Distances are the most suitable heuristics. Other heuristics like h4(n) giving relative position squared weightage do not provide consistent outputs and often do not end up finding the optimal solution.