ASSIGNMENT-2: A* Search

Presented By :-

Ashutosh Samal(2403res15)

List of all user defined Function :-

1- checkValidePath

```
1 usage
def checkValidPath(i,j):   # check if we are not going out of the index
    if i<3 and i>=0 and j<3 and j>=0:
        return True
    else:
        return False
```

This function checks if we are not going out of bound and index are always in the range

2- createChild

```
def createChild(state,i,j,moves): # returns child nodes for a given state and position of blank space
    # i an j is the location of blank space("0")
    childrens=[]

    for move in moves:  # iterate over all the  possible move in moves list.
        if checkValidPath(i+move[0],j+move[1]):
            temp_state=copy.deepcopy(state)
            temp_state[i][j],temp_state[i+move[0]][j+move[1]]=temp_state[i+move[0]][j+move[1]],temp_state[i][j]
            childrens.append(temp_state)
    return childrens
```

This function will take a state, location of Black space as i and j and possible moves. Returns a list of child nodes that can be created from the given state.
Here moves is a list like this [[-1,0], [1,0], [0, -1], [0,1]].

3-MahattenDistance

```
def ManhattanDistance(state,include_blank=False):
    distance=0
    for i in range(3):
        for j in range(3):
            if state[i][j] !=0 or include_blank :
                for x in range(3):
                    for y in range(3):
                        if state[i][j]==final_state[x][y]:
                            distance+=abs(i-x)+abs(y-j)
    return distance
```

Function is used to calculate Manhattan distance of the given state with respect to final state. Part of h3(n) in the given assignment.

This function takes a state and a flag to include blank space or not, default is set to false.
In the if condition if include_blank is true then it will always be true and takes all points in consideration while calculating but in case its false it will not take the Zero/blank pos

4-NoTilesDisplaced

```
def NoTilesDisplaced(state,include_blank=False):
    count=0
    for i in range(3):
        for j in range(3):
            if state[i][j]!=final_state[i][j] and ((state[i][j]!=0 and final_state[i][j]!=0) or include_blank):
                count+=1
    return count
```

Function calculates no of tiles misplaced from its original location its part of h2(n) in the given assignment.

Like Manhattan function this function takes a state and a flag to include blank space or not, default is set to false.

5- ManhattanDistanceWithrandomInt

```
1 usage
def ManhattanDistanceWithrandomInt(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            for x in range(3):
                for y in range(3):
                    if state[i][j] == final_state[x][y]:
                        distance += abs(i - x) + abs(y - j)
    random_integer = random.randint(-30,  b: 30)
    return distance + random_integer
```

This function will calculate Manhattan distance but, in the end, add a random integer between -30 to 30 so that it will overestimate for some nodes and make it non monotonic.

6-Heuristics

```
def Heuristics(state,type,include_blank):
    if type==1:
        return 0
    elif type==2:
        return NoTilesDisplaced(state,include_blank)
    elif type==3:
        return  ManhattanDistance(state,include_blank)
    elif type ==4 :
        return CorrectlyPlacedTiles(state)
    else:
        print("No such Hurestic")
```

I have tried to create a function like factory method which will return different heuristic functions depending upon the parameter type. So that in the main code we can just pass the type of heuristics we want to use.

Its takes state, type and a flag include_blank as parameter and returns the heuristic value of the state as per the given type.

7- getBlankPos

```
2 usages
def getBlankPos(state): # Returns the l
    for i in range(len(state)):
        for j in range(len(state[0])):
            if state[i][j] == 0:
                return [i,j]
```

It's a helper function to return blank position of the given state which will be used by other functions.

8- PrintHueristicType

```
def PrintHueristicType(heuristics_type):
    if heuristics_type==1:
        print("calculating for heuristics h(n)=0")
    elif heuristics_type==2:
        print("calculating for no of misplaced tiles")
    elif heuristics_type==3:
        print("calculating for manhatten distance")
    elif heuristics_type ==4 :
        print("calculating for no of correctly placed tiles")
    else:
        print("No such Hurestic")
```

This is also a helper function to print the type of heuristics being used as per the user input. This will help us to get a good and readable output.

9-Astar:-

```python
def Astar(random_state,final_state,heuristics_type,include_blank=False): # DFS function
    visited = []  # List to store all the visited state
    heap = []  # data structure to be used for A star to store the state to be visited
    moves = [[-1,0], [1,0], [0, -1], [0,1]]  # possible moves
    heap.append([0+Heuristics(random_state,heuristics_type,include_blank),random_state,0]) # Insert Initial  cost,intial state, initial depth
    heapq.heapify(heap)

    while(len(heap)):
        cost,current_state,depth=heapq.heappop(heap) # visited node with least cost

        if final_state == current_state: # check if we have reached target state , If yes return depth
            return [depth,visited,heap]  ## returning depth of solution(depth), visited nodes(visited) and to be visited nodes(heap)
        visited.append(current_state) # add to the visited after visiting the node
        blanck_i,blanck_j=getBlankPos(current_state) # get the position of the blank space
        childerens = createChild(current_state, blanck_i, blanck_j,moves) # get a list of child node from current state which is not visted before
        for child in childerens:
            if child not in visited:
                heapq.heappush( heap,[Heuristics(child,heuristics_type,include_blank)+depth+1,child,depth+1]) # Add all the child node to visit in the stack after checking if we have visited i
```

This is the function to compute A star algo.
Explanation: -
Input:- random_state (It's the initial state to start the computation)
final_state(It's the final state to stop our execution)
heuristics_type (type of heuristics could be 1,2,3 and 4)
Initialized following variables: -
Visited: - stores the states that have been visited

Heap: - Stores the states that have been discovered but not head visited. I have converted it to a min heap so that when I pop from this then I will get the state with minimum cost.

Moves: - Possible moves for the blank space

Before the loop starts insert a list with flowing value in the heap
[initial cost, initial state, initial depth]

initial cost=g(n)+h(n), for start node g(n)=0 and h(n) can be calculated from Heuristics function. I have used this as 1st parameter because heap will sort according to the 1st element in case of multidimension list. And we want to visit the node with minimum cost.

initial state – It's given.

initial depth=0 for the initial node

Code logic: -

- Loop while the len(heap) is not zero. Basically, loop till we don't have any nodes to visit.
- Pop from the queue to get the node with minimum cost to visit.

- Visit this node. i.e check if it’s the goal state, if true return depth, visited list and heap (to be visited list) If not then append it to the visited list.
- If current state is not goal, then expand the child nodes and add then to heap (to be visited list) after checking if they are present already present in visited list as we don’t want to revisit a node.
- Repeat this as long as we don’t get a solution or have visited all the possible nodes.

Tasks in the assignment: -

Task 1: -

We have to run Astar with all the given Heuristics. To run Astar function I have created another helper function.

```python
def RunAstar(initial_state,final_state,heuristics_type,include_blank=False):
    PrintHueristicType(heuristics_type)


    start = time.time()  # Inital timestamp
    result_depth, result_visited,result_tobe_visited = Astar(initial_state, final_state,heuristics_type,include_blank)
    end = time.time()
    print("number of node visited is :-", len(result_visited), " and target is found at a depth of :-", result_depth)
    print("Time taken in secs  to complete the code:- ", end - start)
```

This function will Print number of nodes visited, time taken by the code and depth of the solution. Here depth will be the shortest path length.

For h1(n), in my code heuristics_type will be equal to 1

Output: -

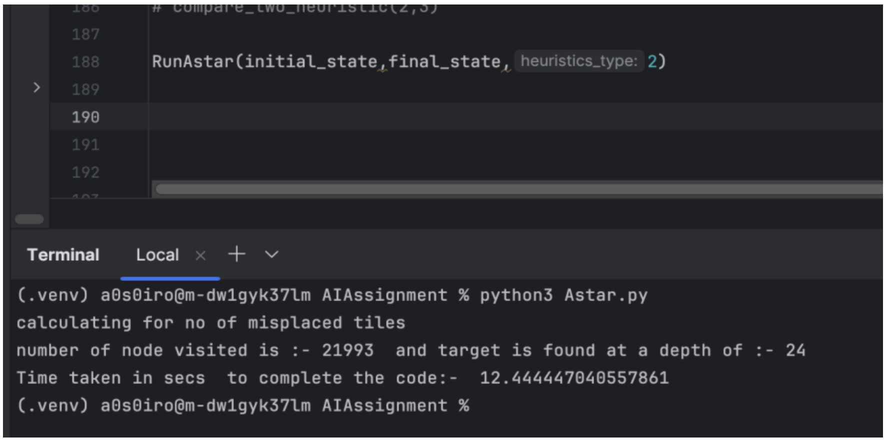
```
RunAstar(initial_state,final_state, heuristics_type: 1)

checkMonotone()
Terminal    Local
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for heuristics h(n)=0
number of node visited is :- 234142  and target is found at a depth of :- 24
Time taken in secs  to complete the code:-  2216.2342340946198
```

Number of nodes visited: - 234142.
Solution depth: - 24 (optimal)
Time taken: - 2216 sec.

For h2(n), in my code heuristics_type will be equal to 2.

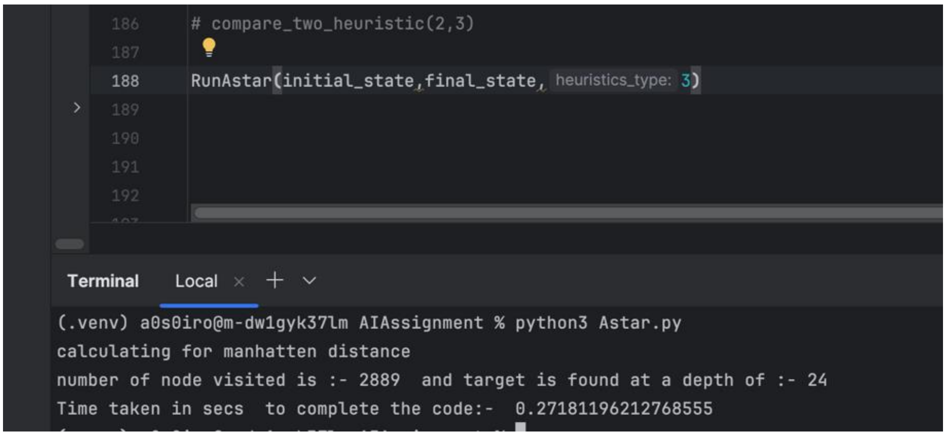
Output: -



Number of nodes visited: - 21993.
Solution depth: - 24 (optimal)
Time taken: - 12 sec.

For h3(n), in my code heuristics_type will be equal to 3.

Output: -



Number of nodes visited: - 2889.
Solution depth: - 24 (optimal)
Time taken: - 0.27 Sec.

For h4(n), in my code heuristics_type will be equal to 4.

Output: -

```
RunAstar(initial_state,final_state, heuristics_type: 4)
```

```
Terminal    Local
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for manhatten distance with random int added
number of node visited is :- 6450  and target is found at a depth of :- 28
Time taken in secs  to complete the code:-  1.255526065826416
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for manhatten distance with random int added
number of node visited is :- 25920  and target is found at a depth of :- 26
Time taken in secs  to complete the code:-  18.874409914016724
```

Number of nodes visited: - 6450, 25920.
Solution depth: - 28,26 (Both are not optimal path)
Time taken: - 1.2 Sec or 18.8 Sec

It gives different results for each run as we are taking random int with Manhattan distance and clearly the path is not optimal.

Task 2: -

To compare 2 heuristics, I have written a function.

```
def compareTwoHeuristic(heuristics_type_1,heuristics_type_2):

    #part of question 1 , 2 and 3
    print("H1 is :-")
    PrintHueristicType(heuristics_type_1)

    print("H2 is :-")
    PrintHueristicType(heuristics_type_2)


    output1_depth,output1_visited,output1_to_be_visted=Astar(initial_state, final_state, heuristics_type_1)

    print("Total node visited in H1 is", len(output1_visited))

    output2_depth,output2_vistied,output2_to_be_visted=Astar(initial_state, final_state, heuristics_type_2)
    #check if type2 is subset of type1
    print("total node visited in H2 is ", len(output2_vistied))


    print("States which are present in good Heuristics but not in bad Heuristics are:-")
    for i in range(min(len(output1_visited),len(output2_vistied))):
        if output2_vistied[i] not in output1_visited and output2_vistied[i] not in  [row[1] for row in output1_to_be_visted] :
            print(output2_vistied[i],i)
```

This function takes 2 heuristics type as input and returns their comparison which are.

1- No of nodes visited by two heuristics.

2- Check if good heuristics is subset of visited node + explored but not yet visited node by bad heuristics.

Output:-

```
compareTwoHeuristic( heuristics_type_1: 2, heuristics_type_2: 3)
```

Terminal   Local

```
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
H1 is :-
calculating for no of misplaced tiles
H2 is :-
calculating for manhatten distance
Total node visited in H1 is 21993
total node visited in H2 is  2889
States which are present in good Heuristics but not in bad Heuristics are:-
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment %
```

Here there is no such nodes which are present in good heuristics but not there in visited + explored nodes in bad heuristics.
Also, we can see the node visited by two heuristics.

Note: - 1st heuristics type in the function should be a bad heuristics compare to other

Task3: -

As we can see from previous task that goal state is reachable in all four heuristics but not optimal for h4(n) as its non-monotonic.

Task4: -

To check monotonic restriction for h2(n) and h3(n) I have written a function to check if

h(n) <= cost(n,m) + h(m)

here I have taken n as any given node and m as its child node. Now to reach its child node c(n,m)=1 ,it can reach child node in just one step

h(n) and h(m) can be calculated using heuristics function

```python
def checkMonotone(initial_state,final_state,heuristics_type,include_blank):
    PrintHueristicType(heuristics_type)

    visited = []  # List to store all the visited state
    heap = []  # data structure to be used for A star to store the state to be visited
    moves = [[-1, 0], [1, 0], [0, -1], [0, 1]]  # possible moves
    heap.append([0 + Heuristics(initial_state, heuristics_type,include_blank), initial_state,0])  # Insert Initial  cost,intial state, initial depth
    heapq.heapify(heap)

    while (len(heap)):
        cost, current_state, depth = heapq.heappop(heap)  # visited node with least cost
        if final_state == current_state:  # check if we have reached target state , If yes return depth
            print("Monotonus Hurestics")
            return [depth, visited,heap]  ## returning depth of solution(depth), visited nodes(visited) and to be visited nodes(heap)
        visited.append(current_state)  # add to the visited after visiting the node
        blanck_i, blanck_j = getBlankPos(current_state)  # get the position of the blank space
        childerens = createChild(current_state, blanck_i, blanck_j,moves)  # get a list of child node from current state which is not visted before
        for child in childerens:
            if child not in visited:
                Hn = Heuristics(current_state, heuristics_type,include_blank)  ## calculating H(n)
                Hm=Heuristics(child, heuristics_type,include_blank)  ## calculating H(m)
                Cn=1 ## C(n,m) will always be 1 as cost from parent to child is always 1
                if Hn>Hm+Cn:
                    print("Given Heuristics is non Monotonus ")
                    print("Nth and Mth node for which H(n)>H(m)+C(n,m) is:-")
                    print("Nth node:-",current_state)
                    print("Mth node:-",child)
                    return

                heapq.heappush(heap, _item: [Heuristics(child, heuristics_type,include_blank) + depth + 1, child,depth + 1])  # Add all the child node to visit in the stack after checking if we ha
```

Wherever the code found a node which does not satisfy the Monotone restriction it will stop the execution and print the Nth and Mth node for which it’s not monotonic.

Checking for h2(n):-

```
checkMonotone(initial_state,final_state, heuristics_type: 2)

ManhattanDistance()

Terminal    Local
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for no of misplaced tiles
Monotonus Hurestics
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment %
```

Checking for h3(n):-

```
checkMonotone(initial_state,final_state, heuristics_type: 3)
```

```
Terminal   Local
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for manhatten distance
Monotonus Hurestics
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment %
```

Checking for h4(n):-

```
checkMonotone(initial_state,final_state, heuristics_type: 4)
```

```
Terminal   Local
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for manhatten distance with random int added
Given Heuristics is non Monotonus
Nth and Mth node for which H(n)>H(m)+C(n,m) is:-
Nth node:- [[3, 2, 1], [4, 5, 0], [8, 7, 6]]
Mth node:- [[3, 2, 1], [4, 0, 5], [8, 7, 6]]
```

Clearly h4(n) is not monotonous so it print the Nth and Mth node for which Monotone restriction is not followed

H2(n) and H3(n) both are monotonic we can also check with intuition that they are never over estimating the cost

h2(n) :- No of tiles not in their correct position , we can see that at least we need 1 step to move a tile which is not in its correct position to its original position

h3(n):- Manhattan distance , it will give the no of tiles between a tile which is not in correct place to its correct place. And we will need at least that many steps to move a tile to its correct position . It never overestimate and greater than h2(n) so it's better.

Task5: -

Now let's try to check monotonicity after including blank space as well in calculating h2(n) and h3(n).

We will use the same function as we used in previous task but with include blank space flag.

H2(n) with blank space: -

```
checkMonotone(initial_state,final_state, heuristics_type: 2, include_blank: True)
```

Terminal   Local

```
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for no of misplaced tiles
Given Heuristics is non Monotonus
Nth and Mth node for which H(n)>H(m)+C(n,m) is:-
Nth node:- [[3, 1, 5], [4, 2, 0], [8, 7, 6]]
Mth node:- [[3, 1, 5], [4, 2, 6], [8, 7, 0]]
```

Goal state is: -
[[1,2,3],
[4,5,6],
[7,8,0]]

Here Nth node is: -
[[3, 1, 5],
[4, 2, 0],
[8, 7, 6]]

H2(n) for this node is: - 8 (Including blank space)

Mth node is: -
[[3, 1, 5],
[4, 2, 6],
[8, 7, 0]]

H2(M) for this node is:- 6(Including blank space)

C(n,m)=1

And we can clearly see that h2(n)>h2(m)+C(n,m). (Not monotonous)

H3(n) with blank space: -

```
checkMonotone(initial_state,final_state, heuristics_type: 3, include_blank: True)
```

```
Terminal    Local
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment % python3 Astar.py
calculating for manhatten distance
Given Heuristics is non Monotonus
Nth and Mth node for which H(n)>H(m)+C(n,m) is:-
Nth node:- [[3, 0, 1], [4, 2, 5], [8, 7, 6]]
Mth node:- [[3, 1, 0], [4, 2, 5], [8, 7, 6]]
(.venv) a0s0iro@m-dw1gyk37lm AIAssignment %
```

Nth node: -
[[3, 0, 1],
[4, 2, 5],
[8, 7, 6]]

Manhattan distance: - 12 (Including blank space)

Mth node: -
[[3, 1, 0],
[4, 2, 5],
[8, 7, 6]]

Manhattan distance: - 10 (Including blank space)

Clearly h3(n)>h3(m)+C(n,m). (Not monotonous)

Conclusion: -

H1(n): - It always returns zero, the cost will always have the backward cost so it's similar to BFS so it will slow as it will expand all the nodes in each depth but it will always give optimal solution .

H2(n): - Number of tiles not in their correct position , we can see that at least we need 1 step to move a tile which is not in its correct position to its original position. SO it's never overestimated which was also proved in above code . And its greater than h1(n) so performed better .

H3(n) :- Manhattan distance , it will give the no of tiles between a tile which is not in correct place to its correct place. And we will need at least that many steps to move a tile to its correct position . So we can conclude with this intuition that it never overestimate and its greater than h2(n) so it's performed best of all the given heuristics

H4(n) :- Manhattan distance with some random integer . It didn’t performed optimally as its not following monotonicity and could overestimate depending upon the random value