CS571 AI LAB 02

04th September 2022

Ishita Singh 1901CS27
Kavya Goyal 1901CS30
Priyanka Sachan 1901CS43

Team Code: 1901cs30_1901cs27_1901cs43
https://colab.research.google.com/drive/1gptoa4_jNoppvMwwYqflY_BV5dpAa73m?usp=sharing

OBJECTIVE

The objective of this assignment was to solve the 8-puzzle problem using different heuristic searches.
The following heuristic searches have been used this assignment
A. g(n) = least cost from source state to current state so far
B. Heuristics

1. h1(n) = 0
2. h2(n) = number of tiles displaced from their destined position
3. h3(n) = sum of Manhattan distance of each tiles from the goal position
4. h4(n) = Devise a heuristics such that h(n) > h* (n).

We derive a heuristic h4 such that:
h4 : Number of tiles out of row + Number of tiles out of column
h4 is an admissible heuristic, since every tile that is out of column or out of row must be moved at least once and every tile that is both out of column and out of row must be moved at least twice.

1. Observe and verify that better heuristics expands lesser states:

After running the observations on input cases, we calculate the number of explored states using the number of elements in the close_list for each algorithm. We arrive at the following h3(n) ≥ h4(n)≥ h2(n) ≥ h1(n)

It is easy to see from the definitions of the given heuristics that, for any node n, h3(n) ≥ h4(n)≥ h2(n) ≥ h1(n). Therefore, heuristic with higher value dominates those with lesser value. Domination translates directly into efficiency.

- ➔ A* using h3 will never expand more nodes than A* using h2 since every node with f(n) < C* will surely be expanded. But because h3 is at least as big as h2 for all nodes, every node that is surely expanded by A* search with h3 will also surely be expanded with h2, and h2 might cause other nodes to be expanded as well. Hence, it is generally better to use a heuristic function with higher values, provided it does not overestimate and that the computation time for the heuristic is not too large.

Similarly, we deduce that the quality of heuristics is in the same order as in h3(n) ≥ h4(n) ≥ h2(n) ≥ h1(n).

```
Enter the start_state: 7B2453681
Enter the target_state: 12345678B
Source State
[['7' 'B' '2']
 ['4' '5' '3']
 ['6' '8' '1']]
Target State
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' 'B']]
```

Algorithm	Optimal Cost	#Optimal Cost States	#Explored States	Execution Time	Reachable	Monotonic
h1(n)	25	26	145402	3.08142	True	True
h2(n)	25	26	28957	0.700285	True	True
h3(n)	25	26	3861	0.110393	True	True
h4(n)	25	26	8656	0.25227	True	True

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

```
# function to check if all states explored by better heuristics are also explored by worse
heuristics
def compareHeuristics(algorithm_better,  close_list_better, algorithm,close_list):
 cnt = 0
 for s in close_list_better:
   if s in close_list:
     cnt = cnt + 1
 if cnt == len(close_list_better):
   print('All the states explored by ' + str(algorithm_better) + ' are explored by ' +
str(algorithm))
 else:
   print('All the states explored by ' + str(algorithm_better) + ' are NOT explored by ' +
str(algorithm))
 return

def compareStates(astar_algorithms):
 for k1 in astar_algorithms:
   for k2 in astar_algorithms:
     if k1 != k2 and k1.algorithm != "h5(n)" and k1.algorithm != "h6(n)" and k2.algorithm != "h5(n)"
and k2.algorithm != "h6(n)":
       compareHeuristics(k1.algorithm, k1.exploredStates,k2.algorithm, k2.exploredStates
```

After a successful result, the results of all the algorithms are compared. Every better heuristic has its explored states as a subset of not better heuristics. For all the h1, h2, h3 and h4, we find such relations

We observe that the set of explored states by
$h3(n) \subseteq h4(n) \subseteq h2(n) \subseteq h1(n)$.
Hence, all the states expanded by better heuristics are also expanded by inferior heuristics.

```
Enter the start_state: 7B2453681
Enter the target_state: 12345678B
Source State
[['7' 'B' '2']
 ['4' '5' '3']
 ['6' '8' '1']]
Target State
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' 'B']]
```

Algorithm	Optimal Cost	#Optimal Cost States	#Explored States	Execution Time	Reachable	Monotonic	Optimal Path
h1(n)	25	26	145402	3.08142	True	True	7B2453681->7524B3
h2(n)	25	26	28957	0.700285	True	True	7B2453681->72B453
h3(n)	25	26	3861	0.110393	True	True	7B2453681->72B453
h4(n)	25	26	8656	0.25227	True	True	7B2453681->7524B3
h5(n)	25	26	30008	0.74663	True	False	7B2453681->72B453
h6(n)	25	26	3779	0.119641	True	False	7B2453681->72B453

```
All the states explored by h1(n) are NOT explored by h2(n)
All the states explored by h1(n) are NOT explored by h3(n)
All the states explored by h1(n) are NOT explored by h4(n)
All the states explored by h2(n) are explored by h1(n)
All the states explored by h2(n) are NOT explored by h3(n)
All the states explored by h2(n) are NOT explored by h4(n)
All the states explored by h3(n) are explored by h1(n)
All the states explored by h3(n) are explored by h2(n)
All the states explored by h3(n) are explored by h4(n)
All the states explored by h4(n) are explored by h1(n)
All the states explored by h4(n) are explored by h2(n)
All the states explored by h4(n) are NOT explored by h3(n)
```

3. Observe and verify monotone restriction on the heuristics.

We observe that all the heuristics are monotonic in nature. We know that

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

Therefore, in order to prove that monotonicity exists, we check for

h(n) > cost(n,m) + h(m)

h (u) <= e (u,v) + h (v), for every u,v such that there is an edge between u and v is the condition for monotonicity, where h is the heuristic function, u and v are vertices in the search graph, and the function e gives the edge cost between u and v (The search graph is undirected).

```
# monotonicity condition
if m.h_n > 1 + heuristicValue:
  flag = 1
```

If for any heuristic, the A* is not monotonic, the flag will record it. After observations, we find that all our heuristics, h1(n), h2(n), h3(n) and h4(n) are monotonically bounded as we can see above included in the Monotonic label. All the heuristics: h1, h2, h3 and h4 are admissible

4. Observe un-reachability and provide a proof.

Out of the 9! = 362880 configurations for 8-puzzle problem, only half of the states are such that target configuration is reached. Therefore, there are 181440 states which do not return the final configuration, hence A* search run 181439 times and return a false i.e. unreachable as the result.

```
Enter the start_state: 31B562487
Enter the target_state: 12345678B
Source State
[['3' '1' 'B']
 ['5' '6' '2']
 ['4' '8' '7']]
Target State
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' 'B']]
```

Algorithm	Optimal Cost	#Optimal Cost States	#Explored States	Execution Time	Reachable	Monotonic	Optimal Path
h1(n)	0	0	181440	3.78431	False	True	
h2(n)	0	0	181440	4.04285	False	True	
h3(n)	0	0	181440	4.88005	False	True	
h4(n)	0	0	181440	4.79553	False	True	
h5(n)	0	0	181440	5.05592	False	False	
h6(n)	0	0	181440	5.15868	False	False	

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

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

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

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

The monotonicity remains intact for both h2,h3 heuristic. (Same as question 3)

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

We change h2 and h3 heuristics, adding the cost of empty tile and check the monotonicity of the derived heuristics. Therefore, we define the following

h5(n) = Number of misplaced tiles with the cost of Blank Tile

```python
# Heuristic 5-> h(n)= Tiles displaced including Blank character tile
def h5Heuristic(intermediateCharacters, targetCharacters):
  cnt = 0
  for e in range(len(targetCharacters)):
    if intermediateCharacters[e] != targetCharacters[e]:
      cnt = cnt + 1
  return cnt
```

h6(n) = Manhattan distance with the cost of Blank Tile

```
# Heuristic 6-> h(n)= Manhatten distance including Blank character
def h6Heuristic(intermediateCharacters, targetCharacters, h3Dictionary):
  dis = 0
  for e in range(len(intermediateCharacters)):
      x = int(e / 3)
      y = e % 3
      # Index of intermediateCharacters[e] in target
      _x = h3Dictionary[intermediateCharacters[e]][0]
      _y = h3Dictionary[intermediateCharacters[e]][1]
      dis = dis + abs(x -_x) + abs(y -_y)
  return dis
```

Algorithm	Optimal Cost	#Optimal Cost States	#Explored States	Execution Time	Reachable	Monotonic	O
h1(n)	25	26	145402	3.08142	True	True	7
h2(n)	25	26	28957	0.700285	True	True	7
h3(n)	25	26	3861	0.110393	True	True	7
h4(n)	25	26	8656	0.25227	True	True	7
h5(n)	25	26	30008	0.74663	True	False	7
h6(n)	25	26	3779	0.119641	True	False	7

As we can see, that monotonicity is broken for h5 and h6 when the cost of blank is added

AS PER INSTRUCTIONS:

1) Use of open list and close list can be easily seen here:

```
# Astar function
def AStar(op,gridCharacters, targetCharacters,idx, h3Dictionary = {}):
    # to check monotonicity restriction
    flag = 0
    # for execution time
    start_time = time.time()
    # for optimal path
    parent_list = {}
    # to keep track of visited states
    visitedDict = {}
    # to keep track of discovered but not explored states
    open_list = PriorityQueue()
    # to keep track of explored states
    close_list = []
    source_h = heuristic(op, gridCharacters, targetCharacters, h3Dictionary)

    # to keep track of visited configurations
    visitedDict[gridCharacters] = 1
    open_list.put(Priority_State(gridCharacters, 0, source_h, idx))
    # number of discovered states
    cld = 0

    while not open_list.empty():
      m = open_list.get()
      close_list.append(m.state)
```

Open List maintains a record of all discovered but unexplored states.
Closed List maintains record of all explored states.

2) Both start state and target state are taken from the user. For most of the processing, we have converted the matrix into string since it greatly simplifies the work.

```
Enter the start_state: 7B2453681
Enter the target_state: 12345678B
Source State
[['7' 'B' '2']
 ['4' '5' '3']
 ['6' '8' '1']]
Target State
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' 'B']]
```

3) //Please refer to the attached output
In case of success:
   - Success Message: Reachable: True
   - Start State: Source State
   - Goal State: Target State
   - Total number of states explored: #Explored States
   - Total number of states to optimal path: #Optimal Cost States
   - Optimal Path: Optimal Path
   - Optimal Path Cost: Optimal Cost
   - Time taken for execution: Execution Time

```
Enter the start_state: 7B2453681
Enter the target_state: 12345678B
Source State
[['7' 'B' '2']
 ['4' '5' '3']
 ['6' '8' '1']]
Target State
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' 'B']]
+------------+---------------+-----------------------+--------------------+-----------------+------------+------------+------------------
| Algorithm  |  Optimal Cost |  #Optimal Cost States |  #Explored States  |  Execution Time | Reachable  | Monotonic  | Optimal Path
+============+===============+=======================+====================+=================+============+============+==================
| h1(n)      |            25 |                    26 |             145402 |         3.08142 | True       | True       | 7B2453681->7524B3
+------------+---------------+-----------------------+--------------------+-----------------+------------+------------+------------------
| h2(n)      |            25 |                    26 |              28957 |        0.700285 | True       | True       | 7B2453681->72B453
+------------+---------------+-----------------------+--------------------+-----------------+------------+------------+------------------
| h3(n)      |            25 |                    26 |               3861 |        0.110393 | True       | True       | 7B2453681->72B453
+------------+---------------+-----------------------+--------------------+-----------------+------------+------------+------------------
| h4(n)      |            25 |                    26 |               8656 |         0.25227 | True       | True       | 7B2453681->7524B3
+------------+---------------+-----------------------+--------------------+-----------------+------------+------------+------------------
| h5(n)      |            25 |                    26 |              30008 |         0.74663 | True       | False      | 7B2453681->72B453
+------------+---------------+-----------------------+--------------------+-----------------+------------+------------+------------------
| h6(n)      |            25 |                    26 |               3779 |        0.119641 | True       | False      | 7B2453681->72B453
+------------+---------------+-----------------------+--------------------+-----------------+------------+------------+------------------
All the states explored by h1(n) are NOT explored by h2(n)
All the states explored by h1(n) are NOT explored by h3(n)
All the states explored by h1(n) are NOT explored by h4(n)
All the states explored by h2(n) are explored by h1(n)
All the states explored by h2(n) are NOT explored by h3(n)
All the states explored by h2(n) are NOT explored by h4(n)
All the states explored by h3(n) are explored by h1(n)
All the states explored by h3(n) are explored by h2(n)
All the states explored by h3(n) are explored by h4(n)
All the states explored by h4(n) are explored by h1(n)
All the states explored by h4(n) are explored by h2(n)
All the states explored by h4(n) are NOT explored by h3(n)
```

In case of failure:

- Failure Message: Reachable: False
- Start State: Source State
- Goal State: Target State
- Total number of states explored before termination: Explored State

```
Enter the start_state: 31B562487
Enter the target_state: 12345678B
Source State
[['3' '1' 'B']
 ['5' '6' '2']
 ['4' '8' '7']]
Target State
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' 'B']]
```

Algorithm	Optimal Cost	#Optimal Cost States	#Explored States	Execution Time	Reachable	Monotonic	Optimal Path
h1(n)	0	0	181440	3.78431	False	True	
h2(n)	0	0	181440	4.04285	False	True	
h3(n)	0	0	181440	4.88005	False	True	
h4(n)	0	0	181440	4.79553	False	True	
h5(n)	0	0	181440	5.05592	False	False	
h6(n)	0	0	181440	5.15868	False	False	

4) //As seen above
   - Total number of states explored: Explored States
   - Total number of states on optimal path: Optimal Cost States
   - Optimal path: Optimal Path
   - Optimal Cost of the path: Optimal Cost
   - Total time taken for execution: Execution Time

5) Submitted .ipynb, .py file

6) The heuristics taken are as follows:
   h1: 0
   h2: Maximum Tiles displaced from their target location
   h3: Manhattan distance
   h4: Maximum Tiles that are not in their correct column + Maximum Tiles that are not in their correct row.

   All four of these are admissible as seen above in the report. Based on above discussions we have concluded that:
   h3> h4> h2> h1 (i.e, h3 is the best and h1 the worst of all)
   This is also supported by the fact that the number of explored states are also the least by h3, followed by h4, then h2 and finally h1, and all of the explored states of a better heuristic are also explored by a worse one.

Relaxation based analysis of heuristics has already been done earlier in the report.
Optimal path cost is the same for all heuristics since they are all admissible.
Time Complexities can be seen here:
h1: O(1)

```
# Heuristic 1-> h(n)=0
def h1Heuristic(intermediateCharacters, targetCharacters):
  return 0
```

h2: O(rows*columns)

```
# Heuristic 2-> h(n)= Tiles displaced ignoring Blank character tile
def h2Heuristic(intermediateCharacters, targetCharacters):
  cnt = 0
  for e in range(len(targetCharacters)):
    if intermediateCharacters[e] != 'B' and intermediateCharacters[e] != targetCharacters[e]:
      cnt = cnt + 1
  return cnt
```

h3: O(rows*columns)

```
# Heuristic 3-> h(n)= Manhatten distance ignoring blank character tile
def h3Heuristic(intermediateCharacters, targetCharacters, h3Dictionary):
  dis = 0
  for e in range(len(intermediateCharacters)):
    if intermediateCharacters[e] != 'B':
      x = int(e / 3)
      y = e % 3
      # Index of intermediateCharacters[e] in target
      _x = h3Dictionary[intermediateCharacters[e]][0]
      _y = h3Dictionary[intermediateCharacters[e]][1]
      dis = dis + abs(x -_x) + abs(y -_y)
  return dis
```

h4: O(rows*columns)

```
# Heuristic 4-> h(n)= Number of tiles out of column + Number of tiles out of row
def h4Heuristic(intermediateCharacters, targetCharacters, h3Dictionary):
  cntrow = 0
  cntcol = 0
  for e in range(len(intermediateCharacters)):
    if intermediateCharacters[e] != 'B':
      x = int(e / 3)
      y = e % 3
      # Index of intermediateCharacters[e] in target
      _x = h3Dictionary[intermediateCharacters[e]][0]
      _y = h3Dictionary[intermediateCharacters[e]][1]
      if x != _x:
        cntrow = cntrow + 1
      if y != _y:
        cntcol = cntcol + 1
  return cntrow + cntcol
```