CS571 AI LAB 04

25th September 2022

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

Team Code: 1901cs30_1901cs27_1901cs43
[Colab Notebook Link]

OBJECTIVE

The objective of this assignment was to implement Simulated Annealing for the 8-puzzle problem for two different heuristics.
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) = number of tiles displaced from their destined position
2. h2(n) = sum of Manhattan distance of each tiles from the goal position

"Simulated Annealing (SA) is a generic probabilistic metaheuristic for the global optimization problem of applied mathematics, namely locating a good approximation to the global minimum of a given function in a large search space."

B. Implement a Simulated Annealing Search Algorithm for solving the 8-puzzle problem. Your start and Goal state should be given in A.

Cooling function used:

```
def get_temperature(max_temperature, iteration, choice):
    if choice == 1:
        return max_temperature*(0.95**iteration)
    elif choice == 2:
        return max_temperature/iteration
    elif choice == 3:
        if iteration == 1:
            return max_temperature
        return max_temperature / math.log(iteration)
```

Heuristics used:

```
# Heuristic 1-> h(n)= Tiles displaced ignoring Blank character tile
def h1Heuristic(intermediateCharacters, targetCharacters):
 cnt = 0
 for e in range(len(targetCharacters)):
   if intermediateCharacters[e] != 'B' and intermediateCharacters[e] != targetCharacters[e]:
     cnt = cnt + 1
 return cnt
```

```
# Heuristic 2-> h(n)= Manhatten distance ignoring blank character tile
def h2Heuristic(intermediateCharacters, targetCharacters, h2Dictionary):
 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 = h2Dictionary[intermediateCharacters[e]][0]
     _y = h2Dictionary[intermediateCharacters[e]][1]
     dis = dis + abs(x -_x) + abs(y -_y)
 return dis
```

Simulated Annealing Algorithm:

```
# Simulated Annealing function
def SimulatedAnnealing(op, sourceCharacters, targetCharacters,idx,cooling_function,
h2Dictionary = {}):

   # Parameters for simulated annealing
   max_iterations = 5*10**4
   max_temperature = 5*10**4
   # for final path
   parent_list = {}
   parent_list[sourceCharacters]=sourceCharacters

   # to keep track of visited states
   visitedDict = {}

   # manage current iteration
   current_iteration = 0
    # current state
   currentState = Priority_State(sourceCharacters, 0, heuristic(op, sourceCharacters,
targetCharacters, h2Dictionary), idx)

   # traversing to neighbours
```

```
    while max_iterations > 0:

      # if target reached
      if currentState.state == targetCharacters:
        return parent_list, True, currentState.g_n, currentState

      current_iteration = current_iteration + 1
      max_iterations = max_iterations - 1

      # mark current state visited
      visitedDict[currentState.state] = 1

      temp = get_temperature(max_temperature, current_iteration, cooling_function)
      if temp == 0:
        temp = 1

      current_cost = currentState.h_n

      # find index of current blank tile
      currentBlank = currentState.blank_state

      # holds all valid next neighbours
      validNextStates = traverse(currentBlank, currentState.state)
      # shuffle neighbors
     random.shuffle(validNextStates)

      # finding next neighbour
      nextState = currentState

      for neighbourCharacters,neighbor_blank in validNextStates:
        if visitedDict.get(neighbourCharacters) != None:
          continue
        neighbour_cost = heuristic(op,neighbourCharacters,targetCharacters,h2Dictionary)
        probability=0
        if neighbour_cost < current_cost:
          probability = 1
        else:
          probability =math.e ** (-1*(neighbour_cost - current_cost) / temp)
        r=random.random()
        if r<= probability:
          nextState=Priority_State(neighbourCharacters, currentState.g_n+1,
neighbour_cost,neighbor_blank)
          parent_list[neighbourCharacters]=currentState.state
        break
       currentState=nextState

    return parent_list, False, currentState.g_n, currentState
```

Output:

```
Enter the source_state: 12345678B
Enter the target_state: 4176B2358
Source State
[['1' '2' '3']
 ['4' '5' '6']
 ['7' '8' 'B']]
Target State
[['4' '1' '7']
 ['6' 'B' '2']
 ['3' '5' '8']]
+-------------+--------------------+------------+------------------------+--------------+------------------+-------------+-------------------
| Algorithm   |   Cooling Function |  Path Cost | Final state reached    |  Path States |   Execution Time | Reachable   | Path Traversed
+=============+====================+============+========================+==============+==================+=============+===================
| h1(n)       |                  1 |        798 | 147326B58              |          799 |         0.329392 | False       | 12345678B->12345B78
+-------------+--------------------+------------+------------------------+--------------+------------------+-------------+-------------------
| h1(n)       |                  2 |       2584 | 26385714B              |         2585 |         0.273704 | False       | 12345678B->1234567B
+-------------+--------------------+------------+------------------------+--------------+------------------+-------------+-------------------
| h1(n)       |                  3 |        746 | B54186732              |          747 |         0.296911 | False       | 12345678B->1234567B
+-------------+--------------------+------------+------------------------+--------------+------------------+-------------+-------------------
| h2(n)       |                  1 |        734 | 4176B2358              |          735 |         0.180079 | True        | 12345678B->12345B78
+-------------+--------------------+------------+------------------------+--------------+------------------+-------------+-------------------
| h2(n)       |                  2 |       1578 | 53417682B              |         1579 |         0.289066 | False       | 12345678B->1234567B
+-------------+--------------------+------------+------------------------+--------------+------------------+-------------+-------------------
| h2(n)       |                  3 |       1792 | 214563B78              |         1793 |         0.295853 | False       | 12345678B->12345B78
+-------------+--------------------+------------+------------------------+--------------+------------------+-------------+-------------------
```

F. Constraints to be checked:

a. Check whether the heuristics are admissible.

The given heuristics H1(n) and H2(n) are admissible because they do not overestimate the cost to the goal node.
For h1(n)-> representing the number of misplaced tiles with respect to the goal state. This means that at least H1(n) moves would be needed to get to the goal node
For h2(n)-> representing the sum of displacement in row and column of each element. This means we need to move at least ci columns and ri rows for each element i in the current state.
For both there heuristics, the following is true:
h(n) <= h*(n)
Therefore both H1(n) and H2(n) are admissible as they do not overestimate the estimated cost to the goal node and hence is less than the actual cost to the goal node.

b. What happens if we make a new heuristics h3 (n)= h1 (n) * h2 (n).

This heuristic may not necessarily be admissible. A heuristic h is admissible if h(n) <= h*(n) where h*(n) is the true cost to a nearest goal. We know that h1 and h2 are admissible. So, h1(n) <= h*(n) and h2(n) <= h*(n). Now, h3(n) = h1(n) * h2(n) does not guarantee that h3(n) <= h*(n). Therefore, the admissibility of the heuristic h3(n) cannot be deduced.

c. What happens if you consider the blank tile as another tile?

The Heuristic value would increase because, initially we do not consider the blank as a tile, but now the error associated with blank tile would also be considered. This might affect admissibility.
If we consider now swapping 2 tiles as our move instead of moving a tile to blank location then we can say that both heuristic are not admissible as see this case

Input state

B 2 3
4 5 6
7 8 9

Goal State

2 B 3
4 5 6
7 8 9

Here using both h1 and h2 our cost comes out to be 2, but in reality only 1 swap is required to reach the goal state(there H <= H* does not hold) Hence both are not admissible.

d. What if the search algorithm got stuck into the Local optimum? Is there any way to get out of this?

In Simulated Annealing, we can get out of the local optimum by accepting candidates with higher cost to escape local optimum. This requires accepting inferior solutions with a certain probability .

e. Compare Hill Climbing (previous assignment) and the Simulated Annealing with respect to optimality, completeness, and running time complexity (only for this specific problem).

Simulated Annealing is an upgrade over Hill Climbing though it can be very computation heavy if it's tasked with many iterations but it is capable of finding a global maximum and not stuck at local optimum. It combines Hill Climbing and Random Walk. Simulated Annealing deals with getting stuck on local optimums by sometimes choosing worse/ suboptimal solutions to get out of the local optimum. It takes worse solutions by assigning them some probability and that helps us to seperate from a greedy paradigm and allows us to move towards some local bad solutions to find the global best solution.Therefore, in terms of completeness Simulated Annealing (SA) performs better than HC provided there are sufficient iterations and appropriate cooling function.
In terms of time complexity, SA is computation heavy and hence more time taking. SA is not optimal because it doesn't necessarily choose the best path on each run.
