ai_lab2_1801cs36_1801cs37

September 2, 2021

0.0.1 Group Id - 1801cs36_1801cs37

- 1801CS36 - Mahesh Babu
- 1801CS37 - Sriram Pingali

0.0.2 Question - 1

The heuristic h1(n) = **Admissible**.

- In the 8-puzzle problem, each displaced tile must be moved at least once to reach the goal state.
- So, the total number of moves to order the tiles correctly, or the cost to reach the goal state will be greater than or equal to the number of displaced tiles.
- Since, this heuristic is not overestimating the cost of reaching the goal state, it is admissible.

The heuristic h2(n) = **Admissible**.

- Since we can only move one block at a time and in only one of the four directions. The optimal scenario for each block is that it has a clear, unobstructed path to its goal state. This is a Manhattan Distance.
- The rest of the states for a pair of blocks is sub-optimal, meaning it will take more moves than the Manhattan Distance to get the block in the right place. *Thus, this heuristic does not overestimate the cost of reaching the goal state. Therefore it is admissible

0.0.3 Question - 2

Not Necessarily 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.

0.0.4 Question - 3

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 admisibility

0.0.5 Question - 4

We can get out of the local optimum by, - * Simulated Annealing - Accept candidates with higher cost to escape local optimum. * Hill Climbing - Random Walk Hill Climbing / Random Restart hill climbing

[1]:
```python
from queue import PriorityQueue
from dataclasses import dataclass
import random
import time
import math
import fileinput

optimal_path = []
MAX_ITERATIONS = 100000
```

Node Class

[2]:
```python
@dataclass(order=True)
class Node():
    """Node class :-
        state: board configuration(a list)
        depth: Node depth - g(n)
        h1: Number of tiles displaced from their destined position
        h2: Sum of Manhattan distance of each tile from the goal position
        parent: parent of the current node
    """
    def __init__(self, state, depth):
        self.state = state
        self.depth = depth
        self.h1 = None
        self.h2 = None
        self.parent = None
        self.priority = random.randrange(1)

    def calc_heuristic(self, goal_state):
        self.h1 = h1_function(self.state, goal_state)
        self.h2 = h2_function(self.state, goal_state)
        self.h3 = self.h1 * self.h2

    def back_track(self):
        current = self
        count = 0
        optimal_path.clear()
        while (current != None):
            bs = board_state(current.state)
            optimal_path.append(current.state)
            count += 1
            current = current.parent
        return count
```

Heuristics

[3]:
```python
def h1_function(current_state, goal_state):
    """Number of tiles displaced from their destined position."""
    cost=0
    # print(current_state, goal_state)
    for i in range(len(current_state)):
        if current_state[i]!=goal_state[i]:
            cost+=1
    return cost

def h2_function(current_state, goal_state):
    """Sum of Manhattan distance of each tile from the goal position."""
    cost=0
    final_position = cordinates(goal_state)
    temp=board_state(current_state)
    for i in range(3):
        for j in range(3):
            t=temp[i][j]
            xf, yf = final_position[t]
            cost += abs(xf-i)+abs(yf-j)
    return cost
```

Utility Functions

[4]:
```python
def board_state(state):
    """Return 2-d matrix representation"""
    i=0
    temp=[([0]*3) for j in range(3)]
    for row in range(3):
        for col in range(3):
            temp[row][col]=state[i]
            i+=1
    return temp

def display_board(state):
    """Print the board"""
    print("-------------")
    print("| %i | %i | %i |" % (state[0], state[1], state[2]))
    print("-------------")
    print("| %i | %i | %i |" % (state[3], state[4], state[5]))
    print("-------------")
    print("| %i | %i | %i |" % (state[6], state[7], state[8]))
    print("-------------")

def display_start_goal(start, goal):
    """Print start and goal states"""
    print("START STATE              GOAL STATE")
    print("-------------            -------------")
```

```
    print("| %i | %i | %i |             | %i | %i | %i |" % (start[0], start[1], start[2], goal[0], goal[1], goal[2]))
    print("-------------             -------------")
    print("| %i | %i | %i |    ---->    | %i | %i | %i |" % (start[3], start[4], start[5], goal[3], goal[4], goal[5]))
    print("-------------             -------------")
    print("| %i | %i | %i |             | %i | %i | %i |" % (start[6], start[7], start[8], goal[6], goal[7], goal[8]))
    print("-------------             -------------")

def cordinates(state):
    """Return position coordinates in the goal state"""
    cords = [None] * 9
    for index, i  in enumerate(state):
        cords[i] = (index // 3, index % 3)
    return cords

# move the blank title up on the board
def move_up(state):
    new_state=state[:]
    index = new_state.index(0)
    if index not in [0,1,2]:
        temp = new_state[index-3]
        new_state[index-3]=new_state[index]
        new_state[index]=temp
    return new_state

# move the blank title down on the board
def move_down(state):
    new_state=state[:]
    index=new_state.index(0)
    if index not in [6,7,8]:
        temp = new_state[index+3]
        new_state[index+3]=new_state[index]
        new_state[index]=temp
    return new_state

# move the blank title left on the board
def move_left(state):
    new_state = state[:]
    index = new_state.index(0)
    if index not in [0,3,6]:
        temp = new_state[index-1]
        new_state[index-1]=new_state[index]
        new_state[index]=temp
    return new_state
```

```
# move the blank title right on the board
def move_right(state):
    new_state = state[:]
    index = new_state.index(0)
    if index not in [2,5,8]:
        temp = new_state[index+1]
        new_state[index+1] = new_state[index]
        new_state[index]=temp
    return new_state
```

[5]:
```
def expansion(state):
    """Expand nodes along the children"""
    expanded_nodes = []
    expanded_nodes.append(move_up(state))       # moving up
    expanded_nodes.append(move_down(state))     # moving down
    expanded_nodes.append(move_left(state))     # moving left
    expanded_nodes.append(move_right(state))    # moving right
    expanded_nodes = [x for x in expanded_nodes if x]
    return(expanded_nodes)
```

Hill Climbing

[6]:
```
def hillClimbing(start_state, goal_state, heuristic):
    """Cost function: f(n) = h(n). In this algorithm, we push the child nodes
    corresponding to the least heuristic greedily and discard the others.
    Might not result in global optimum path"""
    iterations = 0
    open = PriorityQueue()
    closed = []

    start_node = Node(start_state, 0)
    start_node.calc_heuristic(goal_state)

    open.put((getattr(start_node, heuristic), start_node))

    while (not open.empty()):
        iterations+=1
        if iterations>MAX_ITERATIONS:
            print("Failure: No solution found")
            print("Total number of states explored:\n", len(closed))
            return None
        cost, parent = open.get()
        open.queue.clear()
        closed.append(parent.state)
        if (parent.state == goal_state):
            return (parent, len(closed))
        for i in expansion(parent.state):
```

```
            temp_node = Node(i, parent.depth + 1)
            temp_node.parent = parent

            if (temp_node.state == goal_state):
                return (temp_node, len(closed))
            elif (temp_node.state in closed):
                continue

            temp_node.calc_heuristic(goal_state)
            open.put((getattr(temp_node, heuristic), temp_node))
```

Simulated Annealing

[7]:
```
def linear_strategy(temperature):
    temperature = temperature - 0.01
    return temperature

def random_strategy(temperature):
    temperature = random.uniform(0, 1) * temperature*0.9
    return temperature

def negative_exponential(temperature):
    temperature = math.exp(-1*temperature/5) * temperature
    return temperature

def cooling_function(cooling_strategy, temperature):
    return {
        'linear': linear_strategy(temperature),
        'random': random_strategy(temperature),
        'exponential': negative_exponential(temperature)
    }[cooling_strategy]


def SimulatedAnnealing(start_state, goal_state, heuristic, cooling_strategy):
    """Cost function: f(n) = h(n). In this algorithm, we select the node which␣
↪reduces the
    overall heuristic than the parent. If none such children exist, we randomly␣
↪choose a
    child sampled using the boltzmann function"""
    temperature = 0.9
    iterations = 0
    open = PriorityQueue()
    closed = []

    start_node = Node(start_state, 0)
    start_node.calc_heuristic(goal_state)
```

```python
    open.put((getattr(start_node, heuristic), start_node))
    while (not open.empty()):
        iterations+=1
        if iterations>MAX_ITERATIONS:
            print("Failure: No solution found")
            print("Total number of states explored:", len(closed))
            return None
        cost, parent = open.get()
        open.queue.clear()
        closed.append(parent.state)
        if (parent.state == goal_state):
            return (parent, len(closed))


        temperature = max(cooling_function(cooling_strategy, temperature), 0.01)
        expanded_nodes = []
        for i in expansion(parent.state):
            temp_node = Node(i, parent.depth+1)
            temp_node.parent = parent
            temp_node.calc_heuristic(goal_state)
            if (temp_node.state in closed):
                continue
            expanded_nodes.append(temp_node)

        iterations1=0
        while len(expanded_nodes)!=0:
            iterations1+=1
            if iterations1>MAX_ITERATIONS:
                print("Failure: No solution found")
                print("Total number of states explored:", len(closed))
                return None
            random_index = random.randint(0,len(expanded_nodes)-1)
            temp_node = expanded_nodes[random_index]
            if (temp_node.state == goal_state):
                return (temp_node, len(closed))
            deltaE = getattr(temp_node, heuristic) - cost
            acceptProbability = min(math.exp(-1*deltaE / temperature), 1)
            R = random.random()
            if R <= acceptProbability:
                open.put((getattr(temp_node, heuristic), temp_node))
                break
```

Driver code

[8]:
```python
def main():
    # Read input
    lines = fileinput.FileInput(files = 'input.txt')
```

```
# Define start, goal state
start_state = [int(x) for x in lines[0].split(' ')]
goal_state = [int(x) for x in lines[1].split(' ')]

# Define Temperature, Heuristic
temperature = float(lines[2])
heuristic = lines[3].strip("\n")
print(heuristic)
# Define Cooling Strategy
cooling_strategy = lines[4]
display_start_goal(start_state, goal_state)

# Start Hill Climbing Algorithm
print("\nHill Climb Search:\n")
start_clock = time.time()
val = hillClimbing(start_state, goal_state, heuristic)

# If Valid solution, display it
if val!=None:
    node_1 = val[0]
    expl = val[1]

    print("Success! Solution found")
    hc_execution_time = time.time()-start_clock

    print("Total number of states explored:",expl)
    optimal_cost = node_1.back_track()

    print("Total number of states in optimal path:", optimal_cost)
    print("Optimal Path:")

    optimal_path.reverse()
    for state in optimal_path:
        display_board(state)

    print("Optimal path cost:", optimal_cost)
    print("Time taken for execution (Hill Climbing):", hc_execution_time)

# If invalid, print no solution
else:
    print("No solution found")

print("______________________________________________________\n")

# Start Simulated Annealing Algorithm
print("Simulated Annealing Search")
```

```
    start_clock = time.time()
    val = SimulatedAnnealing(start_state, goal_state, heuristic,␣
↪cooling_strategy)

    # If valid solution found, print it
    if val!=None:
        node_1 = val[0]
        expl = val[1]

        print("Success! Solution found")
        sa_execution_time = time.time()-start_clock

        print("Total number of states explored:",expl)
        optimal_cost = node_1.back_track()

        print("Total number of states in optimal path:", optimal_cost)
        print("Optimal Path:")

        for state in optimal_path:
            display_board(state)

        print("Optimal path cost:", optimal_cost)
        print("Time taken for execution (Simulated Annealing Seach):",␣
↪sa_execution_time)

    # If invalid, print no solution
    else:
        print("No solution found")

if __name__ == "__main__":
    main()
```

```
h2
START STATE                GOAL STATE
-------------              -------------
| 1 | 2 | 3 |              | 1 | 2 | 3 |
-------------              -------------
| 0 | 4 | 5 |     ---->    | 4 | 0 | 5 |
-------------              -------------
| 6 | 7 | 8 |              | 6 | 7 | 8 |
-------------              -------------

Hill Climb Search:

Success! Solution found
Total number of states explored: 1
Total number of states in optimal path: 2
```

```
Optimal Path:
-------------
| 1 | 2 | 3 |
-------------
| 0 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
-------------
| 1 | 2 | 3 |
-------------
| 4 | 0 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Optimal path cost: 2
Time taken for execution (Hill Climbing): 0.00017309188842773438
------------------------------------------------------

Simulated Annealing Search
Success! Solution found
Total number of states explored: 1
Total number of states in optimal path: 2
Optimal Path:
-------------
| 1 | 2 | 3 |
-------------
| 4 | 0 | 5 |
-------------
| 6 | 7 | 8 |
-------------
-------------
| 1 | 2 | 3 |
-------------
| 0 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Optimal path cost: 2
Time taken for execution (Simulated Annealing Seach): 0.0017423629760742188

<ipython-input-8-8778a9dab98f>:6: DeprecationWarning: Support for indexing
FileInput objects is deprecated. Use iterator protocol instead.
  start_state = [int(x) for x in lines[0].split(' ')]
<ipython-input-8-8778a9dab98f>:7: DeprecationWarning: Support for indexing
FileInput objects is deprecated. Use iterator protocol instead.
  goal_state = [int(x) for x in lines[1].split(' ')]
<ipython-input-8-8778a9dab98f>:10: DeprecationWarning: Support for indexing
```

```
FileInput objects is deprecated. Use iterator protocol instead.
  temperature = float(lines[2])
<ipython-input-8-8778a9dab98f>:11: DeprecationWarning: Support for indexing
FileInput objects is deprecated. Use iterator protocol instead.
  heuristic = lines[3].strip("\n")
<ipython-input-8-8778a9dab98f>:14: DeprecationWarning: Support for indexing
FileInput objects is deprecated. Use iterator protocol instead.
  cooling_strategy = lines[4]
```

[9]:

```
# start_state= [6, 7, 3, 8, 4, 2, 5, 0, 1]<br>
# goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
# <br><br>
# start_state= [1, 2, 3, 0, 4, 5, 6, 7, 8]<br>
# goal_state = [7, 3, 2, 5, 4, 8, 1, 0, 6]
# <br><br>
# start_state= [1, 2, 3, 0, 4, 5, 6, 7, 8]<br>
# goal_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]
# <br><br>
# start_state= [1, 2, 3, 0, 4, 5, 6, 7, 8]<br>
# goal_state = [2, 3, 5, 6, 1, 8, 4, 7, 0]
# <br><br>
# start_state= [1, 2, 3, 0, 4, 5, 6, 7, 8]<br>
# goal_state = [1, 2, 3, 4, 5, 0, 6, 7, 8]
# <br><br>
# No Solution <br>
# start_state = [6, 7, 3, 8, 4, 2, 1, 0, 5]<br>
# goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
# <br><br>
# start_state = [2, 8, 1, 4, 6, 3, 7, 5, 0]<br>
# goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]
# <br><br>
# start_state = [2, 8, 3, 1, 0, 4, 7, 6, 5]<br>
# goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]

# <br><br>



# 1 2 3 0 4 5 6 7 8<br>
# 1 2 3 4 0 5 6 7 8<br>
# 0.9<br>
# h2<br>
# linear<br>
```

[ ]: