106119029Assessment1

October 23, 2021

0.1 Roll No. - 106119029

0.2 AI/ML Lab Assessment 1

Google Colab Link

0.3 Define a Basic Class for the Game state for the game.

For both the questions we will be making a generalized class that we will be using for hill climbing and the simmulated annealing.

This has functions such as init, board state, movement and manahattan and hamming distance calculations, for the entire game.

[1]:
```
from time import time
import random
from math import exp

class Block(object):
  """
  Represent state of board in 8 puzzle problem.
  """
  n = 0

  def __init__(self, board, prev_state=None):
    assert len(board) == 9

    self.board = board[:]
    self.prev = prev_state
    self.step = 0
    Block.n += 1

    if self.prev:
        self.step = self.prev.step + 1

  def __eq__(self, other):
    """Check wether two state is equal."""
    return self.board == other.board

  def __hash__(self):
```

```python
    """Return hash code of object.

    Used for comparing elements in set
    """
    h = [0, 0, 0]
    h[0] = self.board[0] << 6 | self.board[1] << 3 | self.board[2]
    h[1] = self.board[3] << 6 | self.board[4] << 3 | self.board[5]
    h[2] = self.board[6] << 6 | self.board[7] << 3 | self.board[8]

    h_val = 0
    for h_i in h:
        h_val = h_val * 31 + h_i

    return h_val

def __str__(self):
    string_list = [str(i) for i in self.board]
    sub_list = (string_list[:3], string_list[3:6], string_list[6:])
    return "\n".join([" ".join(l) for l in sub_list])

def manhattan_distance(self):
    """Return Manhattan distance of state."""
    distance = 0
    goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    for i in range(1, 9):
        xs, ys = self.__i2pos(self.board.index(i))
        xg, yg = self.__i2pos(goal.index(i))
        distance += abs(xs-xg) + abs(ys-yg)
    return distance

def hamming_distance(self):
    """Return Hamming distance of state."""
    distance = 0
    goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    for i in range(9):
        if goal[i] != self.board[i]:
            distance += 1
    return distance

def next(self):
    """Return next states from this state."""
    next_moves = []
    i = self.board.index(0)

    next_moves = (self.shift_up(i), self.shift_down(
        i), self.shift_left(i), self.shift_right(i))
```

```
    return [s for s in next_moves if s]

  def shift_right(self, i):
    x, y = self.__i2pos(i)
    if y < 2:
      right_state = Block(self.board, self)
      right = self.__pos2i(x, y+1)
      right_state.__swap(i, right)
      return right_state

  def shift_left(self, i):
    x, y = self.__i2pos(i)
    if y > 0:
      left_state = Block(self.board, self)
      left = self.__pos2i(x, y - 1)
      left_state.__swap(i, left)
      return left_state

  def shift_up(self, i):
    x, y = self.__i2pos(i)
    if x > 0:
      up_state = Block(self.board, self)
      up = self.__pos2i(x - 1, y)
      up_state.__swap(i, up)
      return up_state

  def shift_down(self, i):
    x, y = self.__i2pos(i)
    if x < 2:
      down_state = Block(self.board, self)
      down = self.__pos2i(x + 1, y)
      down_state.__swap(i, down)
      return down_state

  def __swap(self, i, j):
    self.board[j], self.board[i] = self.board[i], self.board[j]

  def __i2pos(self, index):
    return (int(index / 3), index % 3)

  def __pos2i(self, x, y):
    return x * 3 + y
```

1 Question A. Hill Climbing

A local search algorithm tries to find the optimal solution by exploring the states in local region. Hill climbing is a local search technique which always looks for a better solution in its neighborhood.

[2]:
```
class HCSearcher(object):
  """Searcher that manuplate searching process."""

  def __init__(self, start, goal):
    self.start = start
    self.goal = goal

  def print_path(self, state):
    path = []
    while state:
      path.append(state)
      state = state.prev
    path.reverse()
    print("\n-->\n".join([str(state) for state in path]))

  def hill_climbing(self):
    """Run hill climbing search."""
    stack = [self.start]

    while stack:
      state = stack.pop()
      if state == self.goal:
        self.print_path(state)
        print("Found solution")
        break

      h_val = state.manhattan_distance() + state.hamming_distance()
      next_state = False
      for s in state.next():
        h_val_next = s.manhattan_distance() + s.hamming_distance()
        if h_val_next < h_val:
          next_state = s
          h_val = h_val_next
          stack.append(next_state)
          break

      if not next_state:
        self.print_path(state)
        print("Cannot find solution")
```

[3]:
```
def main():
  print("Search for solution\n")
  start = []
```

```
    # The input file must be called input.txt
    '''
    The default input.txt has
    6 4 2
    1 0 3
    7 5 8
    '''

    '''
    This is how you can read it fron input file as grid if you want.
    For showing in collab we will hardcoding the input.
      for line in open("input.txt").readlines():
          for n in line.split():
              start.append(int(n))
      start = Block(start)
    '''


    start = Block([6, 4, 2, 1, 0, 3, 7, 5, 8])
    goal = Block([1, 2, 3, 4, 5, 6, 7, 8, 0])

    search = HCSearcher(start, goal)

    start_time = time()
    search.hill_climbing()
    end_time = time()
    elapsed = end_time - start_time
    print("Search time: %s" % elapsed)


main()
```

```
Search for solution

6 4 2
1 0 3
7 5 8
-->
6 0 2
1 4 3
7 5 8
-->
0 6 2
1 4 3
7 5 8
-->
1 6 2
```

```
0 4 3
7 5 8
-->
1 6 2
4 0 3
7 5 8
-->
1 0 2
4 6 3
7 5 8
-->
1 2 0
4 6 3
7 5 8
-->
1 2 3
4 6 0
7 5 8
-->
1 2 3
4 0 6
7 5 8
-->
1 2 3
4 5 6
7 0 8
-->
1 2 3
4 5 6
7 8 0
Found solution
Search time: 0.0009322166442871094
```

1.0.1 E. Constraints

1. Yes the Heuristics are admissable.
2. The Heuristics are additive so there wont be any noticable change.
3. If there is no blank tiles, no move can be made as all the possible paths are blocked.
4. Yes, we can increase the jump value,this explores multiple options
5. When this state arises,no solution can exist

2 Question B. Simulated Annealing:

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.

[9]:
```python
class SimulateAnnealingSearcher(object):
    """Searcher that manuplate searching process."""

    def __init__(self, start, goal):
        self.start = start
        self.goal = goal

    def print_path(self, state):
        path = []
        while state:
            path.append(state)
            state = state.prev
        path.reverse()
        print("\n-->\n".join([str(state) for state in path]))

    def simulated_annealing(self, heuristic = "displaced",Temperature = 4000):
        stack = [self.start]

        sch = 0.99

        while stack and Temperature>0:
            Temperature *=sch
            state = stack.pop()
            if state == self.goal:
                self.print_path(state)
                print("Found solution")
                break
            if heuristic == "displaced" :
                h_val = state.hamming_distance()
            elif heuristic == "manhattan":
                h_val = state.manhattan_distance()
            else:
                h_val = state.manhattan_distance() + state.hamming_distance()
            next_state = False
            for s in state.next():
                if heuristic == "displaced" :
                    h_val_next = s.hamming_distance()
                elif heuristic == "manhattan" :
                    h_val_next = s.manhattan_distance()
                else:
                    h_val_next = s.manhattan_distance() + s.hamming_distance()
                if h_val_next < h_val or random.uniform(0, 1) < exp(h_val -␣
 ↪h_val_next / Temperature):
                    next_state = s
                    h_val = h_val_next
                    stack.append(next_state)
                    break
```

```
            if not next_state:
                self.print_path(state)
                print("Cannot find solution")
```

[10]:
```
def main2():
    print("Search for solution\n")

    # The input file must be called input.txt
    '''
    The default input.txt has
    6 4 2
    1 0 3
    7 5 8
    '''

    '''
    This is how you can read it fron input file as grid if you want.
    For showing in collab we will hardcoding the input.
      for line in open("input.txt").readlines():
          for n in line.split():
              start.append(int(n))
      start = Block(start)
    '''

    start = Block([0, 2, 3, 1, 4, 6, 7, 5, 8])
    goal = Block([1, 2, 3, 4, 5, 6, 7, 8, 0])

    search = SimulateAnnealingSearcher(start, goal)

    start_time = time()
    search.simulated_annealing("displaced")
    end_time = time()
    elapsed = end_time - start_time
    print("Search time: %s" % elapsed)
    print ("Number of states explored: %d" % Block.n)
```

2.0.1 E. Constraints

1. Yes the Heuristics are admissable.
2. The Heuristics are additive so there wont be any noticable change.
3. If there is no blank tiles, no move can be made as all the possible paths are blocked.
4. Yes, we can increase the jump value,this explores multiple options

[7]:
```
!cp drive/My Drive/Colab Notebooks/106119029.ipynb ./
```

```
Mounted at /content/drive
```

[ ]:

```

```