[CS152] Assignment1

October 27, 2019

The 8-puzzle

Before you turn in this assignment, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then run the test cells for each of the questions you have answered. Note that a grade of 3 for the A* implementation requires all tests in the "Basic Functionality" section to be passed. The test cells pass if they execute with no errors (i.e. all the assertions are passed).

Make sure you fill in any place that says `YOUR CODE HERE`. Be sure to remove the `raise NotImplementedError()` statements as you implement your code - these are simply there as a reminder if you forget to add code where it's needed.

Question 1

Define your PuzzleNode class below. Ensure that you include all attributes that you need to implement an A* search. If you wish, you can even include member functions, such as a function to generate successor states. Alternatively, you can code up this functionality later in the solvePuzzle function.

```
In [668]: import numpy as np
          import random
          from tabulate import tabulate as tb
          from queue import PriorityQueue


          # DEFINES THE CLASS PUZZLE_NODE:
          class PuzzleNode:

              def __init__(self, state, g, h, parent=None, explored=False):
                  """
                  Provides a structure for performing A* search for n^2-1 puzzles
                  """
                  self.state = state
                  self.parent = parent
                  self.g = g # g(n): the cost to reach the node
                  self.h = h # h(n): estimated cost to get from the node to the goal
                  self.explored = explored

              def __lt__(self, other):
```

```
        """
        Compares the total cost f(n) = g(n) + h(n) between two nodes
        """
        return self.g + self.h < other.g + other.h

    def display(self):
        """
        Displays the board of the puzzle at current state
        """
        return tb(self.state, tablefmt="fancy_grid")
```

Question 2

Define your heuristic functions using the templates below. Ensure that you extend the heuristics list to include all the heuristic functions you implement. Note that state will be given as a list of lists, so ensure your function accepts this format. You may use packages like numpy if you wish within the functions themselves.

In [756]:
```
# HEURISTICS FOR THE A* SEARCH:

# Misplaced tiles heuristic
def h1(state):
    """
    This function returns the number of misplaced tiles
    Input:
    - state: the board state as a list of lists
    Output:
    - count: the number of misplaced tiles
    """
    n, count = len(state), 0
    for i in range(n):
        for j in range(n):
            if state[i][j] != 0 and state[i][j] != i * n + j:
                count += 1
    return count

# Manhattan distance heuristic
def h2(state):
    """
    This function returns the Manhattan distance from the solved state
    Input:
    - state: the board state as a list of lists
    Output:
    - dist: the Manhattan distance from the solved configuration
    """
    n, dist = len(state), 0
    for i in range(n):
        for j in range(n):
            if state[i][j] != 0:
                dist += (abs(i - state[i][j] // n)
                         + abs(j - state[i][j] % n))
    return dist

# Linear conflict heuristic
def h3(state):
    """
    This function returns a heuristic that dominates the Manhattan distance
    Input:
    - state: the board state as a list of lists
    Output:
    - h: the Heuristic distance of the state from its solved configuration
    """
    conflict = 0
    goal = generate_goal_state(state)

    for i in range(len(state)):
        for j in range(len(state)):
            if state[i][j] == 0: continue
            correct = np.where(np.array(state) == 0)[0][0], \
                      np.where(np.array(state) == 0)[1][0]

            if (i, j) == correct: continue
            if i == correct[0]:
                for _ in range(j + 1, len(state)):
                    if state[i][j] > state[i][_]:
                        conflict += 1
            elif j == correct[1]:
                for _ in range(i + 1, len(state)):
                    if state[i][j] > state[_][j]:
                        conflict += 1

    return h1(state) + h2(state) + conflict

heuristics = [h1, h2, h3]
```

```
In [635]: # GENERATING THE NEXT MOVE:
          """
          The function explores the different directions
          for the empty tile to move towards.
          """
          def next_move(state):
              # Finding the coordinates of the blank tile
              row = np.where(np.array(state) == 0)[0][0]
              col = np.where(np.array(state) == 0)[1][0]

              successors_list = []
```

```
        # Moving up if the blank tile is not on the first row
        if row != 0:
            next_state = [i.copy() for i in state]
            next_state[row][col], next_state[row-1][col] = \
            next_state[row-1][col], next_state[row][col]
            successors_list.append(next_state)

        # Moving left if the blank tile is not on the first column
        if col != 0:
            next_state = [i.copy() for i in state]
            next_state[row][col], next_state[row][col-1] = \
            state[row][col-1], state[row][col]
            successors_list.append(next_state)

        # Moving down if the blank tile is not on the last row
        if (row != (len(state)-1)):
            next_state = [i.copy() for i in state]
            next_state[row][col], next_state[row+1][col] = \
            state[row+1][col], state[row][col]
            successors_list.append(next_state)

        # Moving right if the blank tile is not on the last column
        if (col != (len(state)-1)):
            next_state = [i.copy() for i in state]
            next_state[row][col], next_state[row][col+1] = \
            state[row][col+1], state[row][col]
            successors_list.append(next_state)

        return successors_list

    """
    Generates the goal state based on the board's dimensions
    """
    def generate_goal_state(state):
        return [list(range(len(state)**2))[i:i+len(state)]
                for i in range(0, len(state)**2, len(state))]
```

Question 3

Code up your A* search using the SolvePuzzle function within the template below. Please do not modify the function header, otherwise the automated testing will fail. You may define other functions or import packages as needed in this cell or by adding additional cells.

```
In [765]: # SOLVE PUZZLE ALGORITHM:

          def solvePuzzle(state, heuristic, show=False):
              """
              This function should solve the n**2-1 puzzle for any n > 2
```

```
Inputs:
- state: The initial state of the puzzle as a list of lists
- heuristic: a handle to a heuristic function.

Outputs:
- steps: The number of steps to optimally solve the puzzle.
- exp: The number of nodes expanded to reach the solution.
- max_frontier: The maximum size of the frontier.
- opt_path: The optimal path as a list of list of lists.
- err: An error code if state is not of the appropriate structure.
"""

# TESTING THE STRUCTURE OF THE INPUT STATE:
"""
State has a square shape nŒn dimension and
contains all numbers from 0 to (n^2)-1
and n is equal or greater than 3
"""
if any(len(row) != len(state)
       for row in state) or list(range(len(state)**2)) \
                            != sorted(sum(state, [])):
    return 0, 0, 0, None, -1

if len(state) < 3:
    return 0, 0, 0, None, -1


# PUZZLE SOLVABILITY:
"""
Based on the notion of number of inverions:
a pair of tiles that are not in the right order
"""
count = 0 # Counting the number of inversions
tiles = sum(state, []) # Flattens the board
for i in range(len(tiles)-1):
    for j in range(i+1, len(tiles)):
        if tiles[j] and tiles[i] and tiles[i] > tiles[j]:
            count += 1

# Finds the blank tile's row order
row = np.where(np.array(state) == 0)[0][0]+1
"""
If the grid width is odd, then the number
of inversions in a solvable situation is even.
"""
if len(state)%2 != 0 and count%2 != 0:
    return 0, 0, 0, None, -2
```

```
"""
If the grid width is even, board is solvable if and only if
    -1: The number of invertions is odd and the row position of the
        blank tile is even counting from the top.
    -2: The number of invertions is even and the row position of the
        blank tile is odd counting from the top.
"""
if len(state)%2 == 0 and row%2 != 0 and count%2 != 0:
    return 0, 0, 0, None, -2

if len(state)%2 == 0 and row%2 == 0 and count%2 == 0:
    return 0, 0, 0, None, -2


# DEFINING THE GOAL STATE:
goal_state = generate_goal_state(state)


initial_node = PuzzleNode(state, 0, heuristic(state))

# Frontier framed as a priority queue
frontier = PriorityQueue()
frontier.put(initial_node)
exp = 0 # Expansion counter

# Dictionary of the cost database
recorded_cost = {str(initial_node.state): initial_node}


# BEGINS THE A* SEARCH:
while not frontier.empty():
    current_node = frontier.get()
    # Check whether the node was explored
    if current_node.explored:
        continue
    # Check if the node is the goal state
    if current_node.state == goal_state:
        break

    # Generate the next move
    next_states = next_move(current_node.state)

    # Evaluating the next moves
    for state in next_states:
        # Incrementing the cost to reach the node
        cost = current_node.g + 1

        """
```

```
                Cross check to find the optimal cost for the next
                Mark the node as explored
                """
                if str(state) in recorded_cost:
                    if recorded_cost[str(state)].g > cost:
                        recorded_cost[str(state)].explored = True
                    else:
                        continue

                # Generating the child node of the current node
                child = PuzzleNode(state, cost, heuristic(state), current_node)
                # Adding the child node into the frontier
                frontier.put(child)
                # Recording the cost of the child node
                recorded_cost[str(child.state)] = child

            exp += 1 # Increments the expansion counter

        # Compiling the optimal path to reach the goal state
        optimal_solution = [current_node]
        while current_node.parent:
            optimal_solution.append(current_node.parent)
            current_node = current_node.parent

        # Inversing the order of the node (start to goal)
        optimal_path = []
        for i in optimal_solution[::-1]:
            optimal_path.append(i.state)

        # Show the pattern of the optimal solution [Optional]
        if show:
            for node in optimal_solution[::-1]:
                print(node.display())

        return len(optimal_solution)-1 , exp, frontier.qsize() , optimal_path, 0
```

Extension Questions
The extensions can be implemented by modifying the code from Q2-3 above appropriately.

1. Initial state solvability: Modify your SolvePuzzle function code in Q3 to return -2 if an initial state is not solvable to the goal state.
2. Extra heuristic function: Add another heuristic function (e.g. pattern database) that dominates the misplaced tiles and Manhattan distance heuristics to your Q2 code.
3. Memoization: Modify your heuristic function definitions in Q2 by using a Python decorator to speed up heuristic function evaluation

There are test cells provided for extension questions 1 and 2.
Basic Functionality Tests

The cells below contain tests to verify that your code is working properly to be classified as basically functional. Please note that a grade of 3 on #aicoding and #search as applicable for each test requires the test to be successfully passed. If you want to demonstrate some other aspect of your code, then feel free to add additional cells with test code and document what they do.

```
In [766]: ## Test for state not correctly defined

          incorrect_state = [[0,1,2],[2,3,4],[5,6,7]]
          _,_,_,_,err = solvePuzzle(incorrect_state, lambda state: 0)
          assert(err == -1)
```

```
In [767]: ## Heuristic function tests for misplaced tiles and manhattan distance

          # Define the working initial states
          working_initial_states_8_puzzle = ([[2,3,7],[1,8,0],[6,5,4]], [[7,0,8],[4,6,1],[5,3,

          # Test the values returned by the heuristic functions
          h_mt_vals = [7,8,7]
          h_man_vals = [15,17,18]

          for i in range(0,3):
              h_mt = heuristics[0](working_initial_states_8_puzzle[i])
              h_man = heuristics[1](working_initial_states_8_puzzle[i])
              assert(h_mt == h_mt_vals[i])
              assert(h_man == h_man_vals[i])
```

```
In [768]: ## A* Tests for 3 x 3 boards
          ## This test runs A* with both heuristics and ensures that the same optimal number o
          ## with each heuristic.

          # Optimal path to the solution for the first 3 x 3 state
          opt_path_soln = [[[2, 3, 7], [1, 8, 0], [6, 5, 4]], [[2, 3, 7], [1, 8, 4], [6, 5, 0]
                           [[2, 3, 7], [1, 8, 4], [6, 0, 5]], [[2, 3, 7], [1, 0, 4], [6, 8, 5]
                           [[2, 0, 7], [1, 3, 4], [6, 8, 5]], [[0, 2, 7], [1, 3, 4], [6, 8, 5]
                           [[1, 2, 7], [0, 3, 4], [6, 8, 5]], [[1, 2, 7], [3, 0, 4], [6, 8, 5]
                           [[1, 2, 7], [3, 4, 0], [6, 8, 5]], [[1, 2, 0], [3, 4, 7], [6, 8, 5]
                           [[1, 0, 2], [3, 4, 7], [6, 8, 5]], [[1, 4, 2], [3, 0, 7], [6, 8, 5]
                           [[1, 4, 2], [3, 7, 0], [6, 8, 5]], [[1, 4, 2], [3, 7, 5], [6, 8, 0]
                           [[1, 4, 2], [3, 7, 5], [6, 0, 8]], [[1, 4, 2], [3, 0, 5], [6, 7, 8]
                           [[1, 0, 2], [3, 4, 5], [6, 7, 8]], [[0, 1, 2], [3, 4, 5], [6, 7, 8]

          astar_steps = [17, 25, 28]
          for i in range(0,3):
              steps_mt, expansions_mt, _, opt_path_mt, _ = solvePuzzle(working_initial_states_
              steps_man, expansions_man, _, opt_path_man, _ = solvePuzzle(working_initial_stat
              # Test whether the number of optimal steps is correct and the same
              assert(steps_mt == steps_man == astar_steps[i])
              # Test whether or not the manhattan distance dominates the misplaced tiles heuri
```

```
            assert(expansions_man < expansions_mt)
            # For the first state, test that the optimal path is the same
            if i == 0:
                assert(opt_path_mt == opt_path_soln)
```

```
In [769]: ## A* Test for 4 x 4 board
          ## This test runs A* with both heuristics and ensures that the same optimal number o
          ## with each heuristic.

          working_initial_state_15_puzzle = [[1,2,6,3],[0,9,5,7],[4,13,10,11],[8,12,14,15]]
          steps_mt, expansions_mt, _, _, _ = solvePuzzle(working_initial_state_15_puzzle, heur
          steps_man, expansions_man, _, _, _ = solvePuzzle(working_initial_state_15_puzzle, heu
          # Test whether the number of optimal steps is correct and the same
          assert(steps_mt == steps_man == 9)
          # Test whether or not the manhattan distance dominates the misplaced tiles heuristic
          assert(expansions_mt >= expansions_man)
```

```
In [770]: # PRINTING AN EXAMPLE OF THE SOLUTION PATH:
          example = solvePuzzle(working_initial_state_15_puzzle, heuristics[1], show=True)
          print("\nThe solution requires {} steps".format(example[0]),
                "with {} expanded nodes".format(example[1]),
                "yeilding a frontier size of {}".format(example[2]))
```

```
 1   2   6   3

 0   9   5   7

 4  13  10  11

 8  12  14  15


 1   2   6   3

 4   9   5   7

 0  13  10  11

 8  12  14  15


 1   2   6   3

 4   9   5   7

 8  13  10  11
```

```
0  12  14  15

 1   2   6   3

 4   9   5   7

 8  13  10  11

12   0  14  15

 1   2   6   3

 4   9   5   7

 8   0  10  11

12  13  14  15

 1   2   6   3

 4   0   5   7

 8   9  10  11

12  13  14  15

 1   2   6   3

 4   5   0   7

 8   9  10  11

12  13  14  15

 1   2   0   3

 4   5   6   7

 8   9  10  11

12  13  14  15
```

```
  1   0   2   3

  4   5   6   7

  8   9  10  11

 12  13  14  15


  0   1   2   3

  4   5   6   7

  8   9  10  11

 12  13  14  15


The solution requires 9 steps with 9 expanded nodes yeilding a frontier size of 12
```

Extension Tests

The cells below can be used to test the extension questions. Memoization if implemented will be tested on the final submission - you can test it yourself by testing the execution time of the heuristic functions with and without it.

```
In [598]: ## Puzzle solvability test

          unsolvable_initial_state = [[7,5,6],[2,4,3],[8,1,0]]
          _,_,_,_,err = solvePuzzle(unsolvable_initial_state, lambda state: 0)
          assert(err == -2)
```

```
In [759]: ## Extra heuristic function test.
          ## This tests that for all initial conditions, the new heuristic dominates over the 

          dom = 0
          for i in range(0,3):
              steps_new, expansions_new, _, _, _ = solvePuzzle(working_initial_states_8_puzzle
              steps_man, expansions_man, _, _, _ = solvePuzzle(working_initial_states_8_puzzle
              # Test whether the number of optimal steps is correct and the same
              assert(steps_new == steps_man == astar_steps[i])
              # Test whether or not the manhattan distance is dominated by the new heuristic i
              # the number of nodes expanded
              dom = expansions_man - expansions_new
          assert(dom > 0)
```

```
          ---------------------------------------------------------------------------
```

```
    AssertionError                            Traceback (most recent call last)

    <ipython-input-759-527f5e8bbbf2> in <module>
     11     # the number of nodes expanded
     12     dom = expansions_man - expansions_new
---> 13 assert(dom > 0)


    AssertionError:
```

```
In [ ]: ## Memoization test - will be carried out after submission
```