CS152 Assignment 1: 8-Puzzle A* Search

Yoav Rabinovich, October 2018

I collaborated with Josh on much of this assignment, as discussed. Research and workings on solvability and the pattern database were done collaboratively.

```python
# Imports
from queue import PriorityQueue
import numpy as np
from collections import deque, defaultdict
```

```python
# Board state class
class PuzzleNode:
    # Constructor
    def __init__(self,state,fval,gval,parent=None):
      # Store the board state
      self.state = state
      # Flag if pruned, so node is not expanded if a better route is found
      self.pruned = False
      # Parent pointer for path reconstruction
      self.parent = parent
      # G-value, representing cost accumulated so far
      self.gval = gval
      # F-value, representing total cost, G-value + H-value,
      # where the latter is the cost to solution approximated by the heuristic functic
      self.fval = fval

    # To string function
    def __str__(self):
      return str(self.coord)

    # Compare total-cost for PriorityQueue to select lowest cost nodes to explore
    def __lt__(self,other):
      return self.fval < other.fval

# Checks that a state is n by n, containing values 0 through n^2-1
def check_State(state,n):

    # Convert to numpy array
    state = np.array(state)

    # Check that shape is n^2
    if state.shape != (n,n):
      return False

    # Compare contents using sets of unique elements
    state_set = set(state.flatten())
    # A comparison set will contain the elements 0 through n^2-1:
    comparison = set(range(n**2))
    # Check for equivalence
    if state_set-comparison != set():
      return False

    # Check solvability (explanation and implementation below)
    if is_Solvable(state) == False:
      return False
    return True

# Takes state (assuming valid, assuming Numpy array) and returns all possible moves as
def possible_Moves(state):

  # Convert to numpy array
  state = np.array(state)

```

```
  n,_ = state.shape
  moves = []

  # Find coords of blank tile
  blank_pos = np.unravel_index(state.argmin(), state.shape)

  # Check for availability of moves, then create the states and add them to moves
  # Up
  if blank_pos[0]!=0:
    move = np.copy(state)
    move[blank_pos], move[blank_pos[0]-1,blank_pos[1]] = move[blank_pos[0]-1,blank_pos
    moves.append(move)
  # Down
  if blank_pos[0]!=(n-1):
    move = np.copy(state)
    move[blank_pos], move[blank_pos[0]+1,blank_pos[1]] = move[blank_pos[0]+1,blank_pos
    moves.append(move)
  # Left
  if blank_pos[1]!=0:
    move = np.copy(state)
    move[blank_pos], move[blank_pos[0],blank_pos[1]-1] = move[blank_pos[0],blank_pos[1
    moves.append(move)
  # Right
  if blank_pos[1]!=(n-1):
    move = np.copy(state)
    move[blank_pos], move[blank_pos[0],blank_pos[1]+1] = move[blank_pos[0],blank_pos[1
    moves.append(move)

  return moves

# Takes a state along with it's dimension, a heuristic function and a boolean print va
# Runs A* search to find the optimal path from the state to the solved state
# Returns number of moves in optimal path, the max size of A* search frontier, and opt
# stp is a boolean that's set to true to also return the number of steps on the optima
def solvePuzzle(n,state,heuristic,prnt=False):

  # Convert state to Numpy array for easier processing
  # Justification for assuming Numpy arrays in other functions
  state = np.array(state)

  if check_State(state,n) == False:
    return 0,0,-1

  # Start node
  start = PuzzleNode(state,heuristic(state),0)

  # We use a dictionary to store explored nodes and their costs, since we will then lo
  # We convert the array into a hashable string for use in the dictionary
  costs = {state.tostring():start}

  # Frontier, stored as a Priority Queue to maintain ordering of states to explore nex
  frontier = PriorityQueue()
  frontier.put(start)

  # A*
  # Initiate counter for steps
  counter = 0
  # Initiate counter for max frontier size
  frontier_size=0

  # As long as the frontier hasn't been emptied, grab the next move to try
  while frontier.empty() == False:
    # Update max frontier size
    frontier_size = max(frontier.qsize(), frontier_size)
    # Get current node to explore
    current = frontier.get()
    # Skip if pruned
    if current.pruned:
      continue
    # Stop when goal is reached by comparing with solved state
    if np.array_equal(current.state.flatten(),range(n**2)):
      # print("goal reached")
      break

```

```
    # Find possible moves
    moves = possible_Moves(current.state)
    # Expand the node in the orthogonal and diagonal directions
    for m in moves:
      # Converting to string for hashing in dict
      m_string = m.tostring()
      # Check if explored
      if m_string in costs:
          # If so, check if the new cost (plus 1 move cost) is lower
          if costs[m_string].gval > current.gval+1:
              # If it is, prune the state which we'll then delete from the frontier
              costs[m_string].pruned = True
          else:
              # ignore the move, since we've already found a better way to get to this
              continue

      # Apply heuristic to determine hval
      hval = heuristic(m)
      # Construct new node for that move
      new = PuzzleNode(m,current.gval+1+hval,current.gval+1,current)
      # Add node to frontier
      frontier.put(new)
      # Add state to explored set
      costs[m_string] = new

      counter = counter + 1
      # print(current.state, counter, len(moves), frontier.qsize(), frontier_size)

    # Backtracking using parent markers
    btrack = [current.state]
    # Iterate until we reach the starting parentless node, and add nodes to path
    while current.parent != None:
        btrack.append((current.parent).state)
        current = current.parent
    # Reverse backtracking path
    path = btrack[::-1]
    if prnt == True:
      # Printing path
      print("Printing path:")
      for step in path:
        print(step)
      print("Size of largest frontier: " + str(frontier.qsize()))
      print("Steps for optimal path: " + str(len(path)))


  # return results
  return len(path), frontier_size, 0

# Heuristic functions

# Memorization decoration
# Whenever a memorized function is called, memorize is substituted in and first checks
# If it does, it returns the solution, otherwise it runs the original function and stc
def memorize(f):
  solutions={}
  def subst_function(state):
    # Convert to numpy array
    state = np.array(state)
    # Convert to string for lookup
    state_string=state.tostring()
    # If state is found, return solution, otherwise solve and memorize
    if state_string in solutions:
      return solutions[state_string]
    solution = f(state)
    solutions[state_string]=solution
    return solution
  return subst_function


# The displaced tiles heuristic function takes a state
# and returns the number of tiles out of order in that state
@memorize
def displaced_Tiles(state):
  # Convert to numpy array
```

```
  state = np.array(state)
  n, _ = state.shape
  counter = 0
  # Flatten state
  state = state.flatten()
  # Iterate over list to check for out of place tiles
  for tile in range(len(state)):
    if tile != state[tile]:
      counter += 1
  return counter

# The manhatten distance heuristic takes a state
# and returns the sum of distances of each tile from it's goal in each dimension
@memorize
def manhatten_Distance(state):
  n,_ = state.shape
  counter = 0
  # Flatten state
  flat = np.copy(state).flatten()
  # Iterate over list and add up distances
  for tile in range(len(flat)):
    # Row distance
    counter += abs(np.floor(flat[tile]/n) - np.floor(tile/n))
    # Column distance
    counter += abs(flat[tile]%n - tile%n)
  return counter

# Heuristics list
heuristics = [displaced_Tiles,manhatten_Distance]

# Takes state and returns True for solvable or False for unsolvable (justification bel
def is_Solvable(state):
  # Determine parity of dimension, True for even, False for odd
  dim_parity = (state.shape[0]%2==0)
  state=state.flatten()
  count=0
  blank_pos=None
  # Determine inversion count
  for i in range(len(state)):
    for j in range(i+1,len(state)):
      if state[j]==0:
        blank_pos=i
        # Don't count the blank
        continue
      if state[i]>state[j]:
        count+=1
  # If n is even, determine and add row count
  if dim_parity==True:
    row_dist=np.floor(i/n)
    count+=row_dist
  return (count%2==0)

```

```
unsolved_states = [
    [[5,7,6],[2,4,3],[8,1,0]],
    [[7,0,8],[4,6,1],[5,3,2]],
    [[2,3,7],[1,8,0],[6,5,4]]]

for i in range(0, 2):
  for j in range(0, 3):
    %time steps, open_setSize, err = solvePuzzle(3, unsolved_states[j], heuristics[i],
    print(i, steps, open_setSize)
```

```
CPU times: user 5.34 s, sys: 30.9 ms, total: 5.37 s
Wall time: 5.37 s
0 28 20782
CPU times: user 2.28 s, sys: 473 µs, total: 2.28 s
Wall time: 2.28 s
0 25 12330
CPU times: user 49.3 ms, sys: 992 µs, total: 50.3 ms
Wall time: 49.7 ms
0 17 420
CPU times: user 587 ms, sys: 3.72 ms, total: 590 ms
```

Solvability:

To check for solvability, we want to find a quantity that can be calculated for any state, and takes one value for all solvable states and only for them. Since no valid move can make a solvable state into an unsolvable state or vice versa, this quanitity will also be invariant under any such move. This is very similar to the nim number in a nim game, although a suboptimal player can lose the game by allowing the nim number to change.

In *Artificial Intelligence: A Modern Approach* by Russel and Norvig, a reference is made to *Winning Ways for Your Mathematical Plays; Vol. 2* by Berlekamp, Conway, and Guy (Academic Press, London, 1982). Based on that suggestion, we start by looking at the number of inversions required to order the state.

For each flattened state, we compare each tile with the tiles after it. For any tile of smaller value, excluding the blank, we increment our inversion count once. Since our target quantity will take one of two values, we look at the parity of that count.

We now have to look at different scenarios, for odd and even state dimensions, to check that the parity is invariant to any valid move.

For odd n

A move sideways will always be expressed in flattened form as the the blank switching places with an adjacent tile, which cannot affect our inversion count since the blank is not taken into account.

A move up or down will switch the blank with tile that's an even distance $(n-1)$ to either direction. The switched tile can only jump over an even number strictly larger or strictly smaller tiles, resulting in an increment or decrement of 2 to the count, or over a combination of smaller and larger tiles that add up to no increment to the count, or a combination of these. Ultimately, with all moves contributing even increments to the count, parity is conserved.

For even n

We start similarly, with a sideways move contributing nothing to the inversion count. However, an up or down move will skip over n-1 tiles, an odd distance. This will result in similar interactions as in the odd n scenario, only with an added increment to the count that doesn't have a partner to cancel out or add up with, and therefore up and down moves are odd, reversing the parity.

Thankfully this can easily be fixed by adding another quantity to the inversion count, that only changes parity with up and down moves. This is simply the row distance, the distance of the blank tile's current row from the first row. It's easy to see that the parity of the row count is invariant to sideways moves, and reverses with up and down moves. Therefore, the parity of the combined quantity of inversion count and row distance, instead of simply the inversion count, will always be conserved in solvable and unsolvable states for even n.

We therefore implemented a solvability checking function using the above proof as justification, and incorporated it into our validity checking function. Since in a solved state the inversion count and the row count are zero, the invariant is zero and therefore even for all solvable states.

Solvabilization

We can also quickly prove for ourselves that any unsolved state can be changed into a solved state, and vice versa, by switching any two adjacent non-blank tiles.

For both parities of n, a sideways moves will result in an increment or decrement of 1 in the parity count, since either a larger tile rises above a smaller tile, or vice versa. The tiles above or below the pair won't notice a difference.

An up and down move will also not be registered by tiles above or below the switched pair, but will affect the tiles in between. To simplify this double switch, we simply describe it as two instances of the regular, blank tile up or down move:

For odd n, an up or down move skips over a even number of tiles, and so will two such moves, as two evens add up to an even. Therefore, this results in an even change to the inversion count. For even n, an up or down move skips over an odd number of tiles, but two such moves skip over an even amount of tiles, as to odds also add up to an even. This then also results in an even change to the inversion count. The effect of the switches cancel out.

However, when we factor in the effect of the switched tiles on one another, we see the same situation as a sideways move added on top of the switch, again resulting in a total increment or decrement of 1 to the inversion count.

Therefore a switch of adjacent non-blank tiles to any direction always inverses the parity of the conserved quantity, effectively swtiching between solvable and unsolvable states.

```
print(is_Solvable(np.array([[1,0,2],[3,4,5],[6,7,8]])))
```

```
True
```

Pattern Database

Heuristics can be generated by loosening restrictions on the problem at hand, to create a quick way to estimate distance to the solution. A Hammond distance heuristic is solving an N-puzzle where it's possible to move any tile anywhere. A Manhatten distance heuristic is solving an N-puzzle where interactions between tiles are not considered. It preforms better because it estimates the real distance more closely, without overestimating in any case. A better heuristic function will impose more constraints on the heuristic problem, but keep solution complexity reasonable.

A pattern database is a heuristic function that makes use of memorized substates, where interactions within each group are considered, but not interactions between the groups. We subdivide the tiles into groups and compute the number of moves to move them into place without considering non-group members except the blank.

A Manhatten distance heuristic can be thought of, therefore, as a special case of the pattern database with a group for each tile. A special case with a single group would be the complete N-puzzle problem, considering interactions between all tiles. We optimize between 1 and n to find a useful amount of groups.

For each group we select, we walk backwards from the solved state in a breadth-first search, and record new states we find. We then have the minimum amount of steps needed to sort the group, since breadth-first search always finds the optimal cost.

We implemented a pattern database exclusively for 8-puzzles for this assignment, and considered 3 groups (3 tiles, 3 tiles, 2 tiles and the blank).

We ran out of time before the submission deadline, with my code still faulty. I chose to leave it in for your consideration. We got further with Josh's code.

```
# Build pattern database for groups
def build_DB(tiles, lookup_tables):

  key = str(sorted(tiles))
  table = lookup_tables[key]

  # Deque for FIFO queuing of states
  open_set = deque()

  solved_state = np.arange(9).reshape((3,3))
  initial_node = PuzzleNode(solved_state,None,0)
  open_set.append(initial_node)

  # Explored states
  explored = defaultdict(bool)

  steps = 0

  while open_set:
      steps += 1
      current = open_set.popleft()

      # Check for previously explored states
      if explored[current.state.tostring()]:
        continue

      # Convert to a string alternating tile numbers and their position for the group
      state_string = ""
      for tile in tiles:
        index = np.argwhere(current.state.flatten() == tile)[0][0]
        state_string += (str(tile) + str(index))

      group_key = state_string

      # Storing state cost
      if group_key not in table:
        table[group_key] = current.gval

      for next_node in possible_Moves(current.state):
          next_state_string = next_node.tostring()
          if not explored[next_state_string]:
            open_set.append(PuzzleNode(solved_state,None,0))

      explored[current.state.tostring()] = True

pattern_db = {}

# Takes a list of lists representing tile groups
# and returns a pattern database heuristic function for groups
def build_pattern_function(groups):

  # Create the database
  groups_key = str(groups)
  pattern_db[groups_key] = defaultdict(dict)

  # Build a database for each group and add to database
  for group in groups:
    group_key = str(sorted(group))
    build_DB(group, pattern_db[groups_key])

  # Define a heuristic function
  def pattern_heuristic(state):

    hval = 0

    # For each group, look up their state and find pattern from database
    for group in groups:

      # Convert to string code as above
      state_string = ""
      for tile in group:
        index = np.argwhere(state.flatten() == tile)[0][0]
        state_string += (str(tile) + str(index))
```

```

      group_key = state_string

      group_table = pattern_db[groups_key][str(sorted(group))]
      hval += group_table[group_key]

    return hval

  return pattern_heuristic

groups = [[3, 6], [1, 4, 7], [2, 5, 8]]
pattern_heuristic = build_pattern_function(groups)
heuristics.append(pattern_heuristic)
```

```
#Checking Pattern DB
#! NOT COMPLETED
for state in unsolved_states:
  solvePuzzle(3, state, heuristics[2], False)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-158-53b32faf05e5> in <module>()
      1 for state in unsolved_states:
----> 2   solvePuzzle(3, state, heuristics[2], False)

<ipython-input-154-ca198771054e> in solvePuzzle(n, state, heuristic, prnt)
     95
     96   # Start node
---> 97   start = PuzzleNode(state,heuristic(state),0)
     98
     99   # We use a dictionary to store explored nodes and their costs, since we w

<ipython-input-157-9cb4f2376718> in pattern_heuristic(state)
     75
     76       group_table = pattern_db[groups_key][str(sorted(group))]
---> 77       hval += group_table[group_key]
     78
     79     return hval

KeyError: '3562'
```

SEARCH STACK OVERFLOW

Josh and I have also started working on an exciting avenue, of a neural-network-driven heuristic function. We sadly didn't have time to finish it, but we discussed trying to incorporate something along those lines on our own, or as part of another CS152 project if applicable. We are very excited about this, and we'll keep you updated!