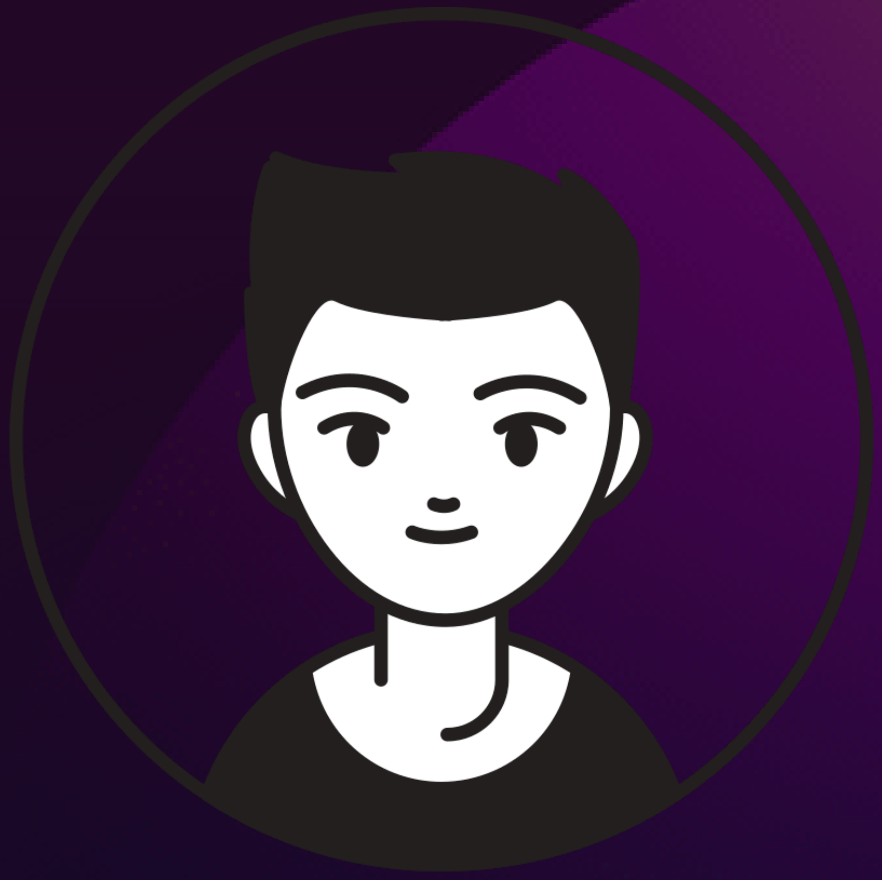
SOLVING 8-PUZZLES WITH INFORMED SEARCH

ARTIFICIAL INTELEGENT



ANDIKA LAKSANA PUTRA
5025211001

M. NAUFAL BAIHAQI
5025211103

INFORMED SEARCH

INFORMED SEARCH

Jenis algoritma pencarian yang menggunakan informasi tambahan (heuristic) tentang masalah yang sedang dicari sehingga dapat mengevaluasi setiap keadaan yang mungkin dilalui dalam mencari solusi

MISPLACED TILES

Teknik lain pada algoritma infromed search dengan menghitung jumlah "Tiles" yang misplaced atau tidak ditempat yang sesuai dengan aturannya. Contohnya adalah ketika memainkan 8-Puzzles, Misplaced Tiles menghitung ada berapa banyak angka yang tidak sesuai dengan tempatnya.

MANHATTAN DISTANCE

Dalam heuristic function, terdapat salah satu teknik sering digunakan dalam penghitungan jarak yaitu Manhattan Distance.Teknik ini menghitung jarak antara dua titik pada bidang koordinat, dengan menghitung selisih antara koordinat x dan koordinat y dari kedua titik tersebut.

$$\text{Manhattan distance} = |x1 - x2| + |y1 - y2|$$

- x1 dan y1 adalah koordinat titik pertama
- x2 dan y2 adalah koordinat titik kedua

```python
import numpy as np
import time

class Node():
    def __init__(self,state,parent,action,depth,step_cost,path_cost,heuristic_cost):
        self.state = state
        self.parent = parent # parent node
        self.action = action # move up, left, down, right
        self.depth = depth # depth of the node in the tree
        self.step_cost = step_cost # g(n), the cost to take the step
        self.path_cost = path_cost # accumulated g(n), the cost to reach the current node
        self.heuristic_cost = heuristic_cost # h(n), cost to reach goal state from the current node

        # children node
        self.move_up = None
        self.move_left = None
        self.move_down = None
        self.move_right = None

    # see if moving down is valid
    def try_move_down(self):
        # index of the empty tile
        zero_index=[i[0] for i in np.where(self.state==0)]
        if zero_index[0] == 0:
            return False
        else:
            up_value = self.state[zero_index[0]-1,zero_index[1]] # value of the upper tile
            new_state = self.state.copy()
            new_state[zero_index[0],zero_index[1]] = up_value
            new_state[zero_index[0]-1,zero_index[1]] = 0
            return new_state,up_value

    # see if moving right is valid
    def try_move_right(self):
        zero_index=[i[0] for i in np.where(self.state==0)]
        if zero_index[1] == 0:
            return False
        else:
            left_value = self.state[zero_index[0],zero_index[1]-1] # value of the left tile
            new_state = self.state.copy()
            new_state[zero_index[0],zero_index[1]] = left_value
            new_state[zero_index[0],zero_index[1]-1] = 0
            return new_state,left_value

    # see if moving up is valid
    def try_move_up(self):
        zero_index=[i[0] for i in np.where(self.state==0)]
        if zero_index[0] == 2:
            return False
        else:
            lower_value = self.state[zero_index[0]+1,zero_index[1]] # value of the lower tile
            new_state = self.state.copy()
            new_state[zero_index[0],zero_index[1]] = lower_value
            new_state[zero_index[0]+1,zero_index[1]] = 0
            return new_state,lower_value

    # see if moving left is valid
    def try_move_left(self):
        zero_index=[i[0] for i in np.where(self.state==0)]
        if zero_index[1] == 2:
            return False
        else:
            right_value = self.state[zero_index[0],zero_index[1]+1] # value of the right tile
            new_state = self.state.copy()
            new_state[zero_index[0],zero_index[1]] = right_value
            new_state[zero_index[0],zero_index[1]+1] = 0
            return new_state,right_value
```

Berisi Constructor dari class Node dan berisi fungsi-fungsi untuk memindahkan posisi 0 pada puzzle

```python
    # return user specified heuristic cost
    def get_h_cost(self,new_state,goal_state,heuristic_function,path_cost,depth):
        if heuristic_function == 'num_misplaced':
            return self.h_misplaced_cost(new_state,goal_state)
        elif heuristic_function == 'manhattan':
            return self.h_manhattan_cost(new_state,goal_state)
        # since this game is made unfair by setting the step cost as the value of the tile being moved
        # to make it fair, I made all the step cost as 1
        # made it a best-first-search with manhattan heuristic function

    # return heuristic cost: number of misplaced tiles
    def h_misplaced_cost(self,new_state,goal_state):
        cost = np.sum(new_state != goal_state)-1 # minus 1 to exclude the empty tile
        if cost > 0:
            return cost
        else:
            return 0 # when all tiles matches

    # return heuristic cost: sum of Manhattan distance to reach the goal state
    def h_manhattan_cost(self,new_state,goal_state):
        current = new_state
        # digit and coordinates they are supposed to be
        goal_position_dic = {1:(0,0),2:(0,1),3:(0,2),8:(1,0),0:(1,1),4:(1,2),7:(2,0),6:(2,1),5:(2,2)}
        sum_manhattan = 0
        for i in range(3):
            for j in range(3):
                if current[i,j] != 0:
                    sum_manhattan += sum(abs(a-b) for a,b in zip((i,j), goal_position_dic[current[i,j]]))
        return sum_manhattan

    # once the goal node is found, trace back to the root node and print out the path
    def print_path(self):
        # create FILO stacks to place the trace
        state_trace = [self.state]
        action_trace = [self.action]
        depth_trace = [self.depth]
        step_cost_trace = [self.step_cost]
        path_cost_trace = [self.path_cost]
        heuristic_cost_trace = [self.heuristic_cost]

        # add node information as tracing back up the tree
        while self.parent:
            self = self.parent

            state_trace.append(self.state)
            action_trace.append(self.action)
            depth_trace.append(self.depth)
            step_cost_trace.append(self.step_cost)
            path_cost_trace.append(self.path_cost)
            heuristic_cost_trace.append(self.heuristic_cost)

        # print out the path
        step_counter = 0
        while state_trace:
            print ('step',step_counter)
            print (state_trace.pop())
            print ('action=',action_trace.pop(),', depth=',str(depth_trace.pop()),\
            ', step cost=',str(step_cost_trace.pop()),', total_cost=',\
            str(path_cost_trace.pop() + heuristic_cost_trace.pop()),'\n')

            step_counter += 1
```

Berisi Fungsi-fungsi yang menjadi ciri khas dari informed search, yaitu get_h_cost, misplaced, manhattan distance, dan juga fungsi untuk print path dari sequence

```python
# search based on path cost + heuristic cost
def a_star_search(self,goal_state,heuristic_function):
    start = time.time()

    queue = [(self,0)] # queue of (found but unvisited nodes, path cost+heuristic cost), ordered by the second element
    queue_num_nodes_popped = 0 # number of nodes popped off the queue, measuring time performance
    queue_max_length = 1 # max number of nodes in the queue, measuring space performance

    depth_queue = [(0,0)] # queue of node depth, (depth, path_cost+heuristic cost)
    path_cost_queue = [(0,0)] # queue for path cost, (path_cost, path_cost+heuristic cost)
    visited = set([]) # record visited states

    while queue:
        # sort queue based on path_cost+heuristic cost, in ascending order
        queue = sorted(queue, key=lambda x: x[1])
        depth_queue = sorted(depth_queue, key=lambda x: x[1])
        path_cost_queue = sorted(path_cost_queue, key=lambda x: x[1])

        # update maximum length of the queue
        if len(queue) > queue_max_length:
            queue_max_length = len(queue)

        current_node = queue.pop(0)[0] # select and remove the first node in the queue

        queue_num_nodes_popped += 1
        current_depth = depth_queue.pop(0)[0] # select and remove the depth for current node
        current_path_cost = path_cost_queue.pop(0)[0] # # select and remove the path cost for reaching current node
        visited.add(tuple(current_node.state.reshape(1,9)[0])) # avoid repeated state, which is represented as a tuple

        # when the goal state is found, trace back to the root node and print out the path
        if np.array_equal(current_node.state,goal_state):
            current_node.print_path()

            print ('Time performance:',str(queue_num_nodes_popped),'nodes popped off the queue.')
            print ('Space performance:', str(queue_max_length),'nodes in the queue at its max.')
            print ('Time spent: %0.2fs' % (time.time()-start))
            return True

        else:
            # see if moving upper tile down is a valid move
            if current_node.try_move_down():
                new_state,up_value = current_node.try_move_down()
                # check if the resulting node is already visited
                if tuple(new_state.reshape(1,9)[0]) not in visited:
                    path_cost=current_path_cost+up_value
                    depth = current_depth+1
                    # get heuristic cost
                    h_cost = self.get_h_cost(new_state,goal_state,heuristic_function,path_cost,depth)
                    # create a new child node
                    total_cost = path_cost+h_cost
                    current_node.move_down = Node(state=new_state,parent=current_node,action='down',depth=depth,\
                                                  step_cost=up_value,path_cost=path_cost,heuristic_cost=h_cost)
                    queue.append((current_node.move_down, total_cost))
                    depth_queue.append((depth, total_cost))
                    path_cost_queue.append((path_cost, total_cost))

            # see if moving left tile to the right is a valid move
            if current_node.try_move_right():
                new_state,left_value = current_node.try_move_right()
                # check if the resulting node is already visited
```

Berisi fungsi dari A*, dalam fungsi ini dapat dipilih menggunakan case misplaced number atau manhattan distance

```python
    # return user specified heuristic cost
    def get_h_cost(self,new_state,goal_state,heuristic_function,path_cost,depth):
        if heuristic_function == 'num_misplaced':
            return self.h_misplaced_cost(new_state,goal_state)
        elif heuristic_function == 'manhattan':
            return self.h_manhattan_cost(new_state,goal_state)
        # since this game is made unfair by setting the step cost as the value of the tile being moved
        # to make it fair, I made all the step cost as 1
        # made it a best-first-search with manhattan heuristic function

    # return heuristic cost: number of misplaced tiles
    def h_misplaced_cost(self,new_state,goal_state):
        cost = np.sum(new_state != goal_state)-1 # minus 1 to exclude the empty tile
        if cost > 0:
            return cost
        else:
            return 0 # when all tiles matches

    # return heuristic cost: sum of Manhattan distance to reach the goal state
    def h_manhattan_cost(self,new_state,goal_state):
        current = new_state
        # digit and coordinates they are supposed to be
        goal_position_dic = {1:(0,0),2:(0,1),3:(0,2),8:(1,0),0:(1,1),4:(1,2),7:(2,0),6:(2,1),5:(2,2)}
        sum_manhattan = 0
        for i in range(3):
            for j in range(3):
                if current[i,j] != 0:
                    sum_manhattan += sum(abs(a-b) for a,b in zip((i,j), goal_position_dic[current[i,j]]))
        return sum_manhattan

    # once the goal node is found, trace back to the root node and print out the path
    def print_path(self):
        # create FILO stacks to place the trace
        state_trace = [self.state]
        action_trace = [self.action]
        depth_trace = [self.depth]
        step_cost_trace = [self.step_cost]
        path_cost_trace = [self.path_cost]
        heuristic_cost_trace = [self.heuristic_cost]

        # add node information as tracing back up the tree
        while self.parent:
            self = self.parent

            state_trace.append(self.state)
            action_trace.append(self.action)
            depth_trace.append(self.depth)
            step_cost_trace.append(self.step_cost)
            path_cost_trace.append(self.path_cost)
            heuristic_cost_trace.append(self.heuristic_cost)

        # print out the path
        step_counter = 0
        while state_trace:
            print ('step',step_counter)
            print (state_trace.pop())
            print ('action=',action_trace.pop(),', depth=',str(depth_trace.pop()),\
            ', step cost=',str(step_cost_trace.pop()),', total_cost=',\
            str(path_cost_trace.pop() + heuristic_cost_trace.pop()),'\n')

            step_counter += 1
```

Berisi Fungsi-fungsi yang menjadi ciri khas dari informed search, yaitu get_h_cost, misplaced, manhattan distance, dan juga fungsi untuk print path dari sequence

```python
# search based on path cost + heuristic cost
def a_star_search(self,goal_state,heuristic_function):
    start = time.time()

    queue = [(self,0)] # queue of (found but unvisited nodes, path cost+heuristic cost), ordered by the second element
    queue_num_nodes_popped = 0 # number of nodes popped off the queue, measuring time performance
    queue_max_length = 1 # max number of nodes in the queue, measuring space performance

    depth_queue = [(0,0)] # queue of node depth, (depth, path_cost+heuristic cost)
    path_cost_queue = [(0,0)] # queue for path cost, (path_cost, path_cost+heuristic cost)
    visited = set([]) # record visited states

    while queue:
        # sort queue based on path_cost+heuristic cost, in ascending order
        queue = sorted(queue, key=lambda x: x[1])
        depth_queue = sorted(depth_queue, key=lambda x: x[1])
        path_cost_queue = sorted(path_cost_queue, key=lambda x: x[1])

        # update maximum length of the queue
        if len(queue) > queue_max_length:
            queue_max_length = len(queue)

        current_node = queue.pop(0)[0] # select and remove the first node in the queue

        queue_num_nodes_popped += 1
        current_depth = depth_queue.pop(0)[0] # select and remove the depth for current node
        current_path_cost = path_cost_queue.pop(0)[0] # # select and remove the path cost for reaching current node
        visited.add(tuple(current_node.state.reshape(1,9)[0])) # avoid repeated state, which is represented as a tuple

        # when the goal state is found, trace back to the root node and print out the path
        if np.array_equal(current_node.state,goal_state):
            current_node.print_path()

            print ('Time performance:',str(queue_num_nodes_popped),'nodes popped off the queue.')
            print ('Space performance:', str(queue_max_length),'nodes in the queue at its max.')
            print ('Time spent: %0.2fs' % (time.time()-start))
            return True

        else:
            # see if moving upper tile down is a valid move
            if current_node.try_move_down():
                new_state,up_value = current_node.try_move_down()
                # check if the resulting node is already visited
                if tuple(new_state.reshape(1,9)[0]) not in visited:
                    path_cost=current_path_cost+up_value
                    depth = current_depth+1
                    # get heuristic cost
                    h_cost = self.get_h_cost(new_state,goal_state,heuristic_function,path_cost,depth)
                    # create a new child node
                    total_cost = path_cost+h_cost
                    current_node.move_down = Node(state=new_state,parent=current_node,action='down',depth=depth,\
                                                  step_cost=up_value,path_cost=path_cost,heuristic_cost=h_cost)
                    queue.append((current_node.move_down, total_cost))
                    depth_queue.append((depth, total_cost))
                    path_cost_queue.append((path_cost, total_cost))

            # see if moving left tile to the right is a valid move
            if current_node.try_move_right():
                new_state,left_value = current_node.try_move_right()
                # check if the resulting node is already visited
```

Berisi fungsi dari A*, dalam fungsi ini dapat dipilih menggunakan case misplaced number atau manhattan distance

```python
# search based on heuristic cost
def best_first_search(self, goal_state):
    start = time.time()

    queue = [(self,0)] # queue of (found but unvisited nodes, heuristic cost), ordered by heuristic cost
    queue_num_nodes_popped = 0 # number of nodes popped off the queue, measuring time performance
    queue_max_length = 1 # max number of nodes in the queue, measuring space performance

    depth_queue = [(0,0)] # queue of node depth, (depth, heuristic cost)
    path_cost_queue = [(0,0)] # queue for path cost, (path_cost, heuristic cost)
    visited = set([]) # record visited states

    while queue:
        # sort queue based on heuristic cost, in ascending order
        queue = sorted(queue, key=lambda x: x[1])
        depth_queue = sorted(depth_queue, key=lambda x: x[1])
        path_cost_queue = sorted(path_cost_queue, key=lambda x: x[1])

        # update maximum length of the queue
        if len(queue) > queue_max_length:
            queue_max_length = len(queue)

        current_node = queue.pop(0)[0] # select and remove the first node in the queue
        #print 'pop'
        #print current_node.state
        #print 'heuristic_cost',current_node.heuristic_cost,'\n'

        queue_num_nodes_popped += 1
        current_depth = depth_queue.pop(0)[0] # select and remove the depth for current node
        current_path_cost = path_cost_queue.pop(0)[0] # # select and remove the path cost for reaching current node
        visited.add(tuple(current_node.state.reshape(1,9)[0])) # avoid repeated state, which is represented as a tuple

        # when the goal state is found, trace back to the root node and print out the path
        if np.array_equal(current_node.state,goal_state):
            current_node.print_path()

            print ('Time performance:',str(queue_num_nodes_popped),'nodes popped off the queue.')
            print ('Space performance:', str(queue_max_length),'nodes in the queue at its max.')
            print ('Time spent: %0.2fs' % (time.time()-start))
            return True

        else:
            # see if moving upper tile down is a valid move
            if current_node.try_move_down():
                new_state,up_value = current_node.try_move_down()
                # check if the resulting node is already visited
                if tuple(new_state.reshape(1,9)[0]) not in visited:
                    # get heuristic cost
                    h_cost = self.h_misplaced_cost(new_state,goal_state)
                    # create a new child node
                    current_node.move_down = Node(state=new_state,parent=current_node,action='down',depth=current_depth+1,\
                                          step_cost=up_value,path_cost=current_path_cost+up_value,heuristic_cost=h_cost)
                    queue.append((current_node.move_down,h_cost))
                    depth_queue.append((current_depth+1,h_cost))
```

Berisikan fungsi BFS yang dikolaborasikan menggunakan metode misplaced number

```python
start = np.array([0,1,3,4,2,5,7,8,6]).reshape(3,3)

initial_state = start
goal_state = np.array([1,2,3,4,5,6,7,8,0]).reshape(3,3)
print (initial_state, '\n')
print (goal_state)

root_node = Node(state=initial_state, parent=None,action=None,depth=0,step_cost=0,path_cost=0,heuristic_cost=0)


""" search based on num_misplaced heuristic cost, using priority queue """
# root_node.best_first_search(goal_state)

""" A*1 search based on path cost+heuristic cost, using priority queue """
# root_node.a_star_search(goal_state,heuristic_function = 'num_misplaced')

""" A*2 search based on path cost+heuristic cost, using priority queue """
root_node.a_star_search(goal_state,heuristic_function = 'manhattan')
```

Berisikan tahap awal dari program. Di tahap ini dipilih ingin menggunakan metode yang mana. Dalam hal ini, sedang digunakan metode A* menggunakan manhattan distance.

Output berisi step-step dari start state ke goal state dan terdapat pula penjelasan dari kedalamannya, step cost nya, serta total cost nya

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

[[1 2 3]
 [4 5 6]
 [7 8 0]]
step 0
[[0 1 3]
 [4 2 5]
 [7 8 6]]
action= None , depth= 0 , step cost= 0 , total_cost= 0

step 1
[[1 0 3]
 [4 2 5]
 [7 8 6]]
action= left , depth= 1 , step cost= 1 , total_cost= 8

step 2
[[1 2 3]
 [4 0 5]
 [7 8 6]]
action= up , depth= 2 , step cost= 2 , total_cost= 9

step 3
[[1 2 3]
 [4 5 0]
 [7 8 6]]
action= left , depth= 3 , step cost= 5 , total_cost= 15

step 4
[[1 2 3]
 [4 5 6]
 [7 8 0]]
action= up , depth= 4 , step cost= 6 , total_cost= 22
```

THANK YOU

I HOPE YOU LEARNED SOMETHING NEW!