RBS

June 18, 2021

[1]:
```
from math import sqrt
import numpy as np

class Grid_5x5:
    def __init__(self):
        self.grid = [
                      [3, 4, 1, 3, 1],
                      [3, 3, 3,'G',2],
                      [3, 1, 2, 2, 3],
                      [4, 2, 3, 3, 3],
                      [4, 1, 4, 3, 2]
                                        ]
        self.goal = 'G'
        self.goal_pos = {"row":1, "col":3}

    #␣
 ↪===========================================================================================
    # Prints 5x5 Grid and also can bold and underline Agent's current state␣
 ↪while printing Grid
    def print_environment(self, current_state=None):

        for r in range(5):
            for c in range(5):

                if current_state:
                    if r == current_state['row'] and c == current_state['col']:
                        # \033[1m is for bold, \033[4m is for underlined,␣
 ↪\033[0m is for finishing both bold and underlined (all)
                        print("\033[1m\033[4m{}\033[0m".format(self.
 ↪grid[r][c]), end=' ')
                    else:
                        print(self.grid[r][c], end=' ')

                else:
                    print(self.grid[r][c], end=' ')

            print()
```

```
        print()
        return

    #␣
↪=============================================================================================
↪
    # Gives the new row after adding any num to it or Gives thhe new column␣
↪after adding any num to it,
    def _increment_pos(self, row_or_col, num_to_move):
        return (row_or_col+num_to_move)%5   # If adding num_to_move to␣
↪row_or_col exceeds 5 (given rows,cols of grid) so thats why using modulo to␣
↪move in circular


# THE NODES LOGIC HERE COULD BE IMPLEMENTED AS GENERAL TOO

#␣
↪=============================================================================================
# Would be used in rbfs searching tree we construct later, f(n) is calculated␣
↪here could also be used for other informed searches i.e A*, greedy etc
class InformedNodeAlternative:
    def __init__(self, parent, state, parent_action, path_cost,␣
↪heuristic_score):
        self.parent = parent
        self.state = state
        self.parent_action = parent_action
        self.path_cost = path_cost    # g(n)
        self.heuristic_score = heuristic_score  # h(n)
        self.f = path_cost + heuristic_score    # f(n) = g(n) + h(n)
#␣
↪=============================================================================================
↪
class InformedChildNodeAlternative(InformedNodeAlternative):
    def __init__(self, problem, parent, parent_action, heuristic_type):
        state = problem.transition_model(parent.state, parent_action)  # This␣
↪will give new state when a state applies an action
        path_cost = parent.path_cost + problem.step_cost(parent.state,␣
↪parent_action)  # This would sum of step costs of path at each individual␣
↪state
        heuristic_score = problem.calculate_heuristic(state, heuristic_type)  #␣
↪calculating heuristic
#         print(parent.heuristic_score, heuristic_score)

        super().__init__(parent=parent,
                         state=state,
                         parent_action=parent_action,
```

```
                         path_cost=path_cost,
                         heuristic_score=heuristic_score
                                                            )# f would be␣
↪calculated in InformedNodeAlternative

#␣
↪==========================================================================================
# According to Book a problem for searching has:
# 1. initial state
# 2. possible actions
# 3. transition model (A description what each action does)
# 4. goal test (which determines that has goal been reached at given state)
# 5. path cost (that assigns numeric cost to each path)

class Problem:
    def __init__(self, Environment, initial_state):
        self.initial_state = initial_state
        self.Environment = Environment
        self.possible_actions = ['horizontal', 'vertical']

    #␣
↪==========================================================================================
↪
    # Gives new state given current state and action applied at current state
    def transition_model(self, current_state, action):
        state, new_state = current_state.copy(), current_state.copy()
        # Note: state/position in grid seemed better to represent as dictionary␣
↪for readibility
        row = state['row']
        col = state['col']
        num_to_move = self.Environment.grid[row][col]

        # if action is to move horizontal then increment the current col of␣
↪state according to current state's value
        if action == 'horizontal':
            new_state['col'] = self.Environment._increment_pos(col, num_to_move)

        # if action is to move vertical then increment the current row of state␣
↪according to current state's value
        elif action == 'vertical':
            new_state['row'] = self.Environment._increment_pos(row, num_to_move)

        return new_state
```

```
    #␣
↪=========================================================================================
↪
    # Tests that whether current node is goal state or not
    def goal_test(self, current_node):
        # print('CHECKING GOAL')
        state = current_node.state
        row = state['row']
        col = state['col']
        value_in_grid = self.Environment.grid[row][col]

#           print('{},{} -> {}'.format(row, col, value_in_grid))
        if value_in_grid == self.Environment.goal:
            return True

        return False

    #␣
↪=========================================================================================
↪
    # step cost of each individual step/state, as there are only two actions␣
↪horizontally and vertically so 1 as step cost for both seems better
    def step_cost(self, current_state, action):
        return 1   # In book assumption is that step costs are non negative

    #␣
↪=========================================================================================
    # calculate euclidean heuristic
    def euclidean_heuristic(self, state):
        goal_pos = self.Environment.goal_pos
        return  sqrt( (state['row']-goal_pos['row'])**2
                                    +
                      (state['col']-goal_pos['col'])**2
                                                          )
    #␣
↪=========================================================================================
↪
    # calculate manhattan heuristic
    def manhattan_heuristic(self, state):
        goal_pos = self.Environment.goal_pos
        return  ( abs( state['row']-goal_pos['row'] )
                                +
                  abs( state['col']-goal_pos['col'] )  )

    #␣
↪=========================================================================================
```

```
    # calculate euclidean heuristic or manhattan heuristic of a state
    def calculate_heuristic(self, state, heuristic_type):
        if heuristic_type == 'euclidean':
            return self.euclidean_heuristic(state)

        elif heuristic_type == 'manhattan':
            return self.manhattan_heuristic(state)

#␣
 ↪===========================================================================================
class GridSearchingAgent():
    def __init__(self, Problem):
#          Problem.Environment.print_environment()
#          Problem.Environment.print_environment(Problem.initial_state)

        self.Environment = Problem.Environment  # seems better
        self.Problem = Problem

    #␣
 ↪===========================================================================================
 ↪
    # Gives sequences of actions from from the branch where goal state was␣
 ↪passed on leaf starting from parent state to leaf node (goal state)
    def actions_to_take(self, current_node):
        if current_node.parent is None:   # base case for recursion
            return []

        return self.actions_to_take(current_node.parent) + [current_node.
 ↪parent_action]

    #␣
 ↪===========================================================================================
    # recursive best first search algorithm, returns a sequence of actions and␣
 ↪performance measure
    def recursive_best_first_search_goal(self, heuristic_type):
        node = InformedNodeAlternative(parent=None,
                                       state=self.Problem.initial_state,
                                       parent_action=None,
                                       path_cost=0,
                                       heuristic_score=problem.
 ↪calculate_heuristic(self.Problem.initial_state, heuristic_type))
```

```
        result, best, search_cost, path_cost = self.RBFS(node, np.inf, heuristic_type, 0)  # np.inf is infinity provided in numpy  also passing heuristic type so it would know which heuristic to compute, 0 is initial search cost, not included in actual algorithm but I have included as it was the trend above too
        # best is used in RBFS below but as its returned here it is unnecessary here
        return result, search_cost, path_cost
    # ==================================================================================================================

    # actual rbfs
    def RBFS(self, node, f_limit, heuristic_type, search_cost):
        search_cost += 1

        # checking goal test on node
        if self.Problem.goal_test(node):
            return self.actions_to_take(node), 0, search_cost, node.path_cost # 0 is immaterial or unimportant as its only for logic to work correctly

        # only creating child nodes of current node
        successors = []
        for action in self.Problem.possible_actions:  # possible_actions(node) but in this problem each state has two possible actions so thats why
            child = InformedChildNodeAlternative(self.Problem, node, action, heuristic_type)
            successors.append(child)

        if len(successors) == 0:
            return None, np.inf, search_cost, node.path_cost   # None, np.inf are used by algorithm

        # setting successor.f to parent's f if its greater
        for successor in successors:
            successor.f = max(successor.path_cost + successor.heuristic_score, node.f)

        while True:
            successors.sort(key=lambda successor: successor.f)  # For priority queue so nodes would be in ascending order of f
            best = successors[0]   # Best node with least f value

            # This means that we need to unwind to alternative path from any ancestor of current node
```

```
            if best.f > f_limit:
                return None, best.f, search_cost, node.path_cost   # None is␣
↪cutoff

            alternative = successors[1].f   # As successors was in ascending␣
↪order of f so after best i.e 0 index, node on 1 index is best as alternative

            result, best.f, search_cost, path_cost = self.RBFS(best,␣
↪min(f_limit, alternative), heuristic_type, search_cost)

            if result is not None:
                return result, best.f, search_cost, path_cost



    #␣
↪==========================================================================================
    # This helper method turns a state {row:x col:y} to (x,y) Note: state/
↪position in grid seemed better to represent as dictionary for readibility␣
↪but for displaying tuple seemed better
    def _state_to_tuple(self, state):
        x = state['row']
        y = state['col']
        return x,y

    #␣
↪==========================================================================================
    # This helper method gives new state (used for printing)
    def _change_state(self, state, action):
        return self.Problem.transition_model(state, action)

    #␣
↪==========================================================================================
    # This helper method displays
    def display_action(self, current_state, action):
        current_pos = self._state_to_tuple(current_state)
        new_state = self._change_state(current_state, action)
        new_pos = self._state_to_tuple(new_state)

        print('Agent moving {} from {} to {}'.format(action, current_pos,␣
↪new_pos))
        self.Environment.print_environment(new_state)

        return new_state

    #␣
↪==========================================================================================
```

```
    # This method will do searching and if solution exists it will also display␣
↪the actions
    def start(self, search_algo, heuristic_type=None):
        print("\n====== {} with h(n)={}========".format(search_algo.upper(),␣
↪heuristic_type))
        print("\n---Agent's initial state is {}---".format( self.
↪_state_to_tuple(self.Problem.initial_state) ) )
        self.Problem.Environment.print_environment(self.Problem.initial_state)

        current_state = self.Problem.initial_state

        # searching for solution
        if search_algo == 'bfs':
            solution = self.breadth_first_search_goal()
        elif search_algo == 'greedy':
            solution = self.greedy_best_first_search_goal(heuristic_type)
        elif search_algo == 'A*':
            solution = self.astar_search_goal(heuristic_type)
        elif search_algo == 'rbfs':
            solution = self.recursive_best_first_search_goal(heuristic_type)

        actions_sequence, search_cost, path_cost = solution

        if actions_sequence:
            for action in actions_sequence:
                current_state = self.display_action(current_state, action)

        print("---Agent has reached 'G' so stopping")
        self.Environment.print_environment(current_state)
        print("search cost:", search_cost)
        print("path cost:", path_cost)
        print("total cost:", search_cost+path_cost)  # total cost combines both␣
↪search cost and path cost
        print('\n')

        return
#␣
↪==========================================================================================
↪

environment = Grid_5x5()
row_input = int(input("Enter the ROW of initial state in 5x5 grid: "))
col_input = int(input("Enter the COL of initial state in 5x5 grid: "))
initial_state = {'row':row_input, 'col':col_input}

problem = Problem(environment, initial_state)
agent = GridSearchingAgent(problem)
```

```
search_algo = 'rbfs'
heuristic_type = input("Enter the heuristic (euclidean or manhattan)?: ")
agent.start(search_algo, heuristic_type)
```

```
Enter the ROW of initial state in 5x5 grid:  1 1 1 1 1
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-acfc3b59ec4d> in <module>
    274
    275 environment = Grid_5x5()
--> 276 row_input = int(input("Enter the ROW of initial state in 5x5 grid: "))
    277 col_input = int(input("Enter the COL of initial state in 5x5 grid: "))
    278 initial_state = {'row':row_input, 'col':col_input}

ValueError: invalid literal for int() with base 10: '1 1 1 1 1 '
```

[ ]: